TAMPERE UNIVERSITY OF TECHNOLOGY

TOMI KURKO

DEVELOPMENT OF A CONTROL FRAMEWORK FOR DRILL TEST BENCHES

Master of Science Thesis

Examiner: Prof. Tommi Mikkonen
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 5th February 2014

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Electrical Engineering
KURKO, TOMI: Development of a control framework for drill test benches
Master of Science Thesis, 58 pages
August 2014
Major: Embedded Systems
Examiner: Professor Tommi Mikkonen
Keywords: Automatic testing, automatic test system, ATS, testing framework, drill test bench, rock drill

Sandvik Mining and Construction Oy manufactures drill rigs, which are machines used for rock drilling. A drill rig has one or more rock drills that perform the actual drilling. The rock drills are tested in test benches, which are automatic test systems. These drill test benches are used mainly for assuring quality of manufactured rock drills.

In the beginning of the thesis process, Sandvik had project plans on building new drill test benches for testing durability of new rock drill products in the product development stage. Durability testing involves long test runs where test phases are repeated several times. The new drill test benches required a new control system to be developed. The control system is an embedded system consisting of several components, some of which run the Linux operating system. Additionally, a desktop application needed to be developed for creating tests for the drill test benches.

In this thesis a concept of a testing framework is created, which describes conceptually how the automatic tests for the drill test benches are defined and run. Applicability of the concept is analyzed also in a broader context for testing various kinds of systems. An implementation of the testing framework concept for drill test benches is developed, which is the main focus of this thesis. All levels of the control system are described briefly from both hardware and software perspectives. However, the thesis will not go into details of other software components beyond the testing framework since they were not implemented by the author.

The project plans for building new drill test benches based on the new control system were cancelled late in the process of writing this thesis. Therefore, no feedback from production use was received. The control system can, however, be deployed later in new drill test bench projects or when an old drill test bench control system is updated. Based on current assessment, the project has achieved its goals. The new software used for creating tests has better usability than the one used in a previous drill test bench. Maintainability of the control system is also considerably better than in previous drill test benches.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Sähkötekniikan koulutusohjelma
KURKO, TOMI: Poratestipenkin ohjausjärjestelmän kehittäminen
Diplomityö, 58 sivua
Elokuu 2014
Pääaine: Sulautetut järjestelmät
Tarkastaja: Professori Tommi Mikkonen
Avainsanat: Automaattinen testaus, automaattinen testijärjestelmä, ATS, testikehys, poratestipenkki, kallioporakone

Sandvik Mining and Construction Oy valmistaa poralaitteita, joita käytetään kallionporaukseen. Ne ovat työkoneita, joissa on yksi tai useampia porakoneita, jotka tekevät varsinaisen porauksen. Porakoneiden testaamiseen käytetään automaattisia testipenkkejä. Poratestipenkkejä käytetään pääasiallisesti valmistettavien porakoneiden laadunvarmistukseen.

Diplomityön aloitushetkellä Sandvikilla oli suunnitteilla useita projekteja, joissa rakennettaisiin testipenkkejä uusien porakonemallien kestotestausta varten. Tuotekehitysvaiheessa porakoneille tehdään pitkiä testiajoja, joissa testivaiheita toistetaan useita kertoja. Uusiin testipenkkeihin tarvittiin uusi ohjausjärjestelmä, jota alettiin kehittää. Ohjausjärjestelmä on sulautettu järjestelmä, jonka laitteisto koostuu useista komponenteista. Osassa komponenteista suoritetaan Linux-käyttöjärjestelmää. Ohjausjärjestelmän lisäksi tarvittiin pöytätietokoneelle toteutettava ohjelmisto, jolla voidaan luoda testejä poratestipenkeille.

Tässä työssä esitellään testikehyksen konsepti, joka kuvaa käsitteellisellä tasolla, miten poratestipenkin testejä luodaan ja suoritetaan. Konseptin soveltuvuutta pohditaan myös laajemmin erilaisten järjestelmien testaamiseen. Tämän työn keskeisenä tavoitteena on kehittää esitetyn konseptin mukainen testikehys poratestipenkkiä varten. Työssä esitellään lyhyesti ohjausjärjestelmän eri tasot sekä laitteiston että ohjelmiston näkökulmasta, mutta testikehyksen ulkopuolisia ohjelmistokomponentteja ei kuvata tarkemmin, sillä niiden toteuttaminen ei ollut projektissa diplomityön tekijän vastuulla.

Työn loppuvaiheilla suunnitelmat uuteen ohjausjärjestelmään pohjautuvien poratestipenkkien rakentamisesta peruuntuivat, joten käyttäjäkokemuksia ja palautetta ohjausjärjestelmän tuotantokäytöstä ei saatu. Ohjausjärjestelmä voidaan kuitenkin ottaa käyttöön myöhemmin toteutettavissa testipenkeissä tai päivitettäessä vanhemman testipenkin ohjausjärjestelmää. Nykyisen arvion mukaan projekti on saavuttanut sille asetetut tavoitteet. Uusi testien luomiseen tarkoitettu ohjelmisto on käytettävyydeltään parempi kuin aiemman testipenkin ohjelmisto. Ohjausjärjestelmän ylläpidettävyys on myös huomattavasti parempi kuin aiemmilla poratestipenkeillä.

PREFACE

This thesis was written while I was working at Bitwise Oy, which is a medium-sized software company located in Tampere, Finland. The thesis was made of a project Bitwise had with its customer Sandvik Mining and Construction Oy.

I would like to thank the supervisor and examiner of this thesis, Professor Tommi Mikkonen, for giving me support during the writing process and providing ideas on how to outline the research topic. I would also like to thank Sandvik and Bitwise for giving me the opportunity to work in this interesting project and use it as a topic for my master's thesis. Especially I would like to thank Tomi Nieminen from Sandvik, who agreed to be interviewed, arranged a tour in the Sandvik's test mine and the factory, and provided advice on where to find more information on the topic. Furthermore, I would like to thank Tomi Mikkonen for letting me use some of my work time for writing the thesis. Finally, I would like to thank all the people who gave me help in proofreading this thesis or supported me in this process by any other means.

Tampere, 24th June 2014

Tomi Kurko
Tieteenkatu 12 A 22
33720 Tampere
tel: +358 45 132 7610
e-mail: tomi.kurko@iki.fi

CONTENTS

ABBREVIATIONS

AC	Alternating Current
API	Application Programming Interface
ATE	Automatic Test Equipment
ATS	Automatic Test System
CAN	Controller Area Network
COM	Component Object Model
CU	Calculation Unit
DA	Data Access
DC	Direct Current
DCOM	Distributed Component Object Model
DCS	Distributed Control System
DTB	Drill Test Bench
DUT	Device Under Test
ECU	Electronic Control Unit
GUI	Graphical User Interface
HMI	Human-Machine Interface
I/O	Input/Output
IC	Integrated Circuit
IEC	International Electrotechnical Commission
MC	Machine Control
MCC	Machine Control C platform
OPC	Open Platform Communications
PC	Personal Computer
PID	Proportional-Integral-Derivative
PLC	Programmable Logic Controller
RRC	Radio Remote Controller
SCADA	Supervisory Control and Data Acquisition
SICA	Sandvik Intelligent Control Architecture
SIL	Safety Integrity Level
SUP	Supervisor
SUT	System Under Test
TiP	Testing in Production
UA	Unified Architecture
USB	Universal Serial Bus
UUT	Unit Under Test
XML	Extensible Markup Language

1. INTRODUCTION

Sandvik is a high-technology, global industrial group offering tools and tooling systems for metal cutting, equipment and tools for the mining and construction industries, and products in advanced stainless steels and special alloys. Sandvik conducts operations within five business areas: Sandvik Mining, Sandvik Construction, Sandvik Machining Solutions, Sandvik Materials Technology, and Sandvik Venture. [31] In Finland Sandvik's mining and construction business areas operate under a company named Sandvik Mining and Construction Oy. Sandvik Mining and Construction offers a wide range of equipment for rock drilling, rock excavation, processing, demolition, and bulk-materials handling. [32] For testing their drill equipment, Sandvik uses *drill test benches (DTB)*, which can simulate various drilling scenarios without performing actual rock drilling.

Drill test benches are *automatic test systems (ATS)*. ATSs are systems that perform tests automatically on a unit that is being tested. In DTBs the unit under test is a rock drill, and it is tested for its mechanical properties. Automatic testing means that a person designs and creates a test beforehand with their desktop computer and loads the test in the DTB, which then automatically runs the test. The person can monitor the operation of the DTB, but this is not necessary since the DTB can monitor itself and stop the test automatically if it seems that the equipment has broken down or some failure has occurred. In addition to the automatic test mode, DTBs can be controlled manually with a radio remote controller or a fixed control panel.

In this thesis a new control system is developed for DTBs used by Sandvik. It is designed specifically for durability testing where test runs can go on for tens of hours. Originally, there were plans to build new DTBs that would utilize the new control system, but these projects were cancelled late in the process of writing this thesis. However, the control system can be deployed later in forthcoming DTB projects or in old DTBs whose control systems need to be updated. The control system is described mainly from the software point of view, and the hardware is presented only briefly. Development of the control system software involves creating a concept of a testing framework. It serves as a conceptual basis for implementing a testing framework that can be used to create and run automatic tests for DTBs. Additionally, new software components are implemented for diagnostics functions and lower level control of the DTB. In the software descriptions the focus is on the testing framework that was implemented by the author.

The purpose of the testing framework concept is to recognize the key ideas that are used to solve the customer's problem and present them in a generalized manner. It is learned in this thesis that many components of the DTB software can be implemented in a way that makes the implementation free of domain specific information. A major contributor to this is the use of signal based testing methodology. Other ideas of the concept are built on top of this idea. A testing framework similar to the presented concept, called ConTest, has been developed by the ABB company, which uses it to automatically test industrial controller applications [11]. Applicability of the concept to various testing scenarios and its main characteristics are analyzed to evaluate the usefulness of the concept in solving other testing related problems as well.

Despite of the presented generalization, the main purpose of this thesis and the project is to solve the customer's problem in a way that best suits the customer's needs and also efficiently utilizes existing components and solutions. Sandvik has been developing its own software platform and system architecture called SICA (Sandvik Intelligent Control Architecture) for several years. SICA is the basis used for most application projects Sandvik starts today, and it was chosen to be used in the DTB project as well. SICA is not only a software platform but also a system architecture and a library of supported hardware components, which implies that many technology choices are fixed when SICA is selected for an application project.

The drill test bench architecture consists of several hardware and software components. First of all, the tests need to be created with a tool which is run in a desktop computer. This graphical tool takes ideas from a tool used in a previous DTB, but it is created from scratch. The DTB in this project has a display which is used for monitoring the DTB and controlling the automatic test mode. The display software utilizes the SICA SUP (Supervisor) platform, which provides a GUI (Graphical User Interface) framework and many services. New views are implemented for the SUP display for showing diagnostics, adjusting parameters, and collecting test data, and the tool for editing tests is made available from the SUP display as well. A software component that runs the tests, the test engine, is implemented for the SUP module. Another component is implemented for logging test data. Although the SUP level is responsible for running the tests, the actual control of the DTB is done by the MC (Machine Control) level. The MC level utilizes the SICA MCC platform (Machine Control C platform), which provides a framework for running applications written in C language. Applications that run controllers and monitor measurements are implemented for the MCC.

All new software components that are needed for the new control system are described in this thesis. The focus is, however, on explaining in more detail only those components that the author was responsible for. This includes the test editor tool, the test engine, and some views for the SUP display. The MC level implementation and the data logging and diagnostics functionalities for the SUP display were implemented by the customer, and more detailed description of them is left outside the scope of this thesis.

The rest of this thesis is structured as follows. Chapter 2 describes some theoretical background of testing in general. Chapter 3 presents the testing framework concept, which forms a conceptual basis for the implemented solution for the customer's problem. Chapter 4 describes the design and implementation of the test editor tool and depicts its user interface by showing screenshots. Chapter 5 discusses DTBs in more detail and describes the system architecture and the human-machine interface. Chapter 6 describes the design and implementation of the test engine and depicts the user interface for running tests from the SUP display. In Chapter 7 the testing framework concept and the success of the implementation are evaluated. Finally, Chapter 8 summarizes the results of this thesis and draws conclusions.

2. TESTING

Testing is a crucial part of any software or hardware development project. According to Myers et al. it was a well-known rule of thumb, already in 1979, that “in a typical programming project approximately 50 percent of the elapsed time and more than 50 percent of the total cost were expended in testing the program or system being developed”. Today, more than a third of a century later, the same holds true. Despite advances in testing and development tools, it seems that this fact is not going to change anytime soon. [18, p. ix]

Software and hardware have become increasingly complex, and testing complex systems requires increasingly more resources. Despite numerous well-known examples of projects that have failed due to insufficient testing, the importance of testing and the time needed for sufficient testing is still commonly underestimated. This chapter gives an overview of different aspects to testing. First, differences in testing software, hardware, and embedded systems are outlined in Section 2.1. There are different scopes of testing which are related to different kinds of requirements imposed on a system. These scopes are discussed in Section 2.2. A hierarchy of testing levels, applicable to testing software and embedded systems, is described in Section 2.3. Moreover, different testing strategies can be used, which are presented in Section 2.4. Since testing is time-consuming, efficiency of testing is important, and it can be improved with test automation. Benefits of automatic testing are discussed in Section 2.5. Finally, terminology related to automatic test systems is described in Section 2.6.

2.1 Software, hardware, and embedded systems testing

Testing can be classified into three fields: software testing, hardware testing and embedded systems testing. Software is always run on some hardware, thus making software dependent on correct behaviour of hardware. In software testing the focus is on verifying and validating the behaviour of software, and the hardware is assumed to work as intended. Hardware testing is defined here as testing any electrical device which is not an embedded system. An embedded system is a system which has been designed for a specific purpose and involves tight co-operation of software and computer hardware to accomplish the desired functionality of the system. Each of these fields of testing have their own characteristics. Delving into the details of each field is beyond the scope of this thesis, but short overviews are presented in the following paragraphs.

In general, testing of a product can be divided into *product development testing* and *production testing*. Product development testing aims to provide quality information for the development team in order to guarantee sufficient quality of a finished product. Production testing, on the other hand, aims to validate that a product has been assembled and configured correctly, functions as designed, and is free from any significant defects. [4, p. 1]

Software is used in various contexts, and it can be classified into at least the following categories: desktop PC (personal computer) applications, server applications, mobile applications, web applications, and software run in embedded systems. Despite different characteristics between these applications, the same testing methodologies can be applied to all of them, including a hierarchy of testing levels, and different testing strategies. Software testing mainly focuses on verifying and validating the design of a piece of software. Software is not manufactured in the traditional sense like hardware, so production testing of software is somewhat different. According to van't Veer, production testing, or *Testing in Production (TiP)*, is a group of test activities that use the diversity of the production environment and real end user data to test the behaviour of software in the live environment [34]. Need for TiP depends on the type of the developed application. A web application developed for a specific customer for a specific purpose may need to be tested in production separately for each installation of the application. Desktop and mobile applications, on the other hand, are usually tested once per each supported platform, and the application is expected to work with all instances of the same platform.

Examples of hardware include various printed circuit boards and integrated circuits (IC). Testing methods and needs vary depending on the case in question. Testing is done in several stages, but the levels known from software testing are not applicable to these systems. In general, hardware testing can be divided into product development testing and production testing. Production testing plays an important role in guaranteeing appropriate quality of manufactured units.

Embedded systems range from simple systems, such as an alarm clock, to complex real-time control systems. For instance, machine and vehicle systems are currently *distributed control systems (DCS)*. Software and hardware of embedded systems can be tested separately to some extent, but the system needs to be tested as a whole as well, where the level hierarchy of software testing can be applied to some extent. Embedded systems are tested both during the product development and in the production phase. Objectives of tests in these phases are different, however. In product development, tests are thorough involving all features of the system. In production testing, only *smoke tests* might be performed, which test only the major functions of the system [26].

2.2 Scopes of testing

Systems have various kinds of requirements that need to be addressed in testing. Most obvious are the functional requirements, but systems have also non-functional requirements that need to be tested. This section describes some scopes of testing following the classification presented in The Art of Software Testing in Chapter 6 [18]. Many more categories related to software testing exist, but they are irrelevant in the scope of this thesis and are therefore left out.

Functional testing aims to verify that a system acts according to its specification. It focuses on verifying the functional behaviour of a system. *Performance testing* concentrates on verifying performance and efficiency objectives such as response times and throughput rates [18, p. 126]. In testing mechanical and hydraulic hardware, several other performance characteristics might be verified. *Stress testing* subjects a system to heavy loads or stresses [18, p. 123]. It is used to determine the stability of a system. Stress testing involves testing a system beyond normal operational capacity to find possible breaking points. *Reliability testing* aims to ensure that the quality and durability of a system is consistent with its specifications throughout the system's intended lifecycle [22].

2.3 Levels of testing

Testing can be done at several levels or stages. The IEEE Std. 829-1998 Standard for Software Test Documentation identifies four levels of test: Unit, Integration, System, and Acceptance. [6, p. 54]

Unit testing is a process of testing individual units, such as software modules or hardware components, of a system. Rather than testing the system as a whole, testing is focused on the smaller building blocks of the system. This makes it easier to find the cause of an error since the error is known to exist in the *unit under test (UUT)*, which should be considerably smaller system than the whole system. Unit testing also facilitates parallel testing since it allows one to test multiple units simultaneously. [18, p. 85]

Integration testing aims to ensure that the various units of a system interact correctly and function cohesively. Integration and integration testing can be performed at various levels depending on the structural complexity of the system. Integration testing yields information on how the units of a system work together, especially at the interfaces. [6, p. 130]

In *system testing* a system is tested as a whole with all units integrated. The purpose of system testing is to compare the system to its specification and original objectives. Myers et al. describe yet another level of testing, *function testing*, which is here regarded as belonging to system testing and is therefore not discussed further. [18, pp. 119-120]

Acceptance testing is the highest level of testing, and it is usually performed by the customer or end user of a system. It is a process of comparing the system to its requirements and the needs of its end users. [18, p. 131]

These levels of testing follow the classic V-model of software development (see [6, p. 101] for more information). They cannot be deployed to hardware or embedded systems testing as such, but they can be used as a basis for classifying testing to different levels. For instance, a modified V-model can be used in DCS production testing [4, pp. 55-57]. Ahola describes how it is used in production testing of mining machines. The model is described in the following paragraphs. It includes four levels of test: Unit tests, Module tests, Functional tests, and System validation.

Unit tests aim to detect low-level assembly faults right after an electric sub-assembly is completed. This involves testing electrical connections and simple electrical devices, like switches, fuses, and relays. Unit tests should mostly be automated. At this stage, CANopen devices are also initially programmed and configured.

Module tests aim to detect faults that are introduced to the system when several smaller sub-assemblies are integrated. One module may contain dozens of electric wires, several CAN (Controller Area Network) bus components, and hydraulic actuators. The module is verified against the design specification. A tester system which simulates the DCS components that are not yet connected to the module is needed for module tests. After the module is verified, rough calibrations are made.

Functional tests aim to verify the correct functionality of the integrated DCS. All functions that can be tested outside of a test mine are tested at this stage. The tests are based on the functional specification of the machine.

System validation aims to validate the whole control system, including all automatic functions. Testing is conducted in a real working environment in a test mine, or in test benches. Final calibrations are made at this stage.

2.4 Testing strategies

Software testing recognizes three different testing strategies: *black-box testing*, *white-box testing*, and *gray-box testing*. The main difference in the ideology between these strategies is the amount of information that is available to the tester on the internal workings of the *system under test (SUT)*. The different testing strategies are illustrated in Figure 2.1.

Black-box testing treats the SUT as a “black box” whose behaviour is verified by observing its outputs, which are the result of given inputs. Testing is done without any reference to internals of the system, that is, the internal implementation and internal state. In other words, only the public interface of the system is known, and no consideration is given to how the system is implemented internally. This approach, however, has a major weakness in that software might have a special handling for particular inputs, which may easily go untested since the implementation details are unknown. Finding all errors in the

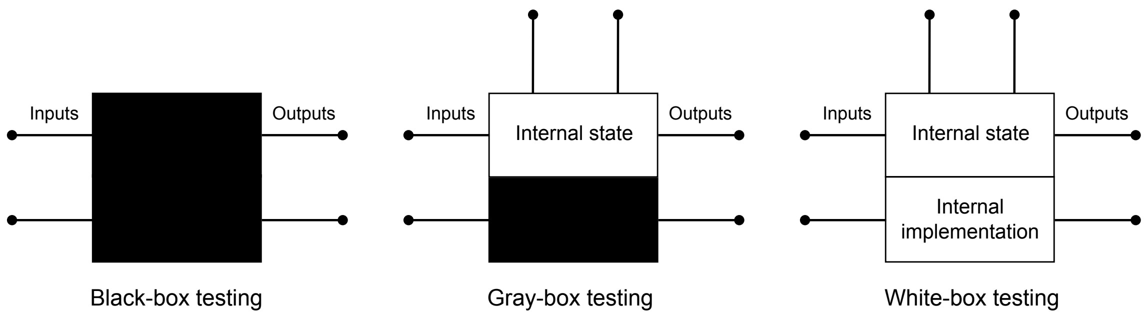


Figure 2.1. *Different testing strategies: black-box, gray-box, and white-box testing. The figure has been modified from the figure in source [14].*

software would require testing with all possible inputs. However, exhaustive input testing is impossible because, in most cases, it would require an infinite number of test cases. [6, p. 159]; [11]; [14]; [18, pp. 9-10]

White-box testing verifies the external behaviour of software as well, but, additionally, it verifies that the internal behaviour is correct. In some cases the software might produce a correct result even if its implementation is incorrect. White-box testing aims to find errors more effectively by examining the internal structure and logic of the software and deriving test data from the examination. This requires complete access to the software's source code. White-box testing also requires the testers to be able to read software design documents and the code. [6, pp. 160-161]; [11]; [18, p. 10]

Gray-box testing is a combination of black-box and white-box testing strategies. Gray-box testing is mostly similar to black-box testing, but it adds the capability to access the internal state of the SUT. Gray-box testing can be used when it is necessary to manipulate internal state such as initial conditions, states, or parameters. [11]; [14]

The terminology originates from software testing and is therefore most meaningful in that context. However, the idea behind the categorization can be seen applicable to some extent to hardware testing as well. When only the public interface of hardware is accessible, the testing can be referred to as black-box testing. If some internal state is accessible as well, the testing is gray-box testing. An example of how internal state can be exposed in electric circuits is the usage of *test points* [23]. With test points, test signals can be transmitted into and out of printed circuit boards. Testing where test data would be obtained from an examination of the internal implementation of the SUT could be seen as white-box testing.

2.5 Motivation for automatic testing

Testing needs to be effective at finding as many defects as possible to gain confidence that the system works as intended. Most of the requirements can be verified by manual testing provided that enough resources are available. However, since testing is very time-consuming and resources are limited, test automation can be used to make testing more efficient.

Automatic testing can significantly reduce the effort required for adequate testing, or increase the amount of testing that can be done with the limited resources. Especially in software testing, tests that would take hours to run manually can be run in minutes. In some cases automating software testing has resulted in savings as high as 80% of manual testing effort. In some other cases automatic testing has not saved money or effort directly, but it has enabled a software company to produce better quality software more quickly than would have been possible by manual testing alone. [7, p. 3] The following paragraphs describe some of the benefits of automatic testing over manual testing [7, pp. 9-10].

Efficiency: Automatic testing reduces the time needed to run tests. The amount of speed-up depends on the SUT and the testing tools. Reduction in run time makes it possible to run more tests and more frequently, which leads to greater confidence in the system and is likely to increase the quality of the system. Automatic testing also results in better use of human resources. Automating repetitive and tedious tasks frees skilled testers' time, allowing them to put more effort into designing better test cases. Moreover, when there is considerably less manual testing, the testers can do the remaining manual testing better.

Repeatibility: Automated tests make it easy to run existing tests on new versions of a system, which allows regression testing to be done efficiently. The test runs are consistent since they are repeated exactly the same way each time. The same tests can also be executed in different environments, such as with different hardware configurations. This might be economically infeasible to perform by manual testing.

Reusability: When designed well, automated tests can easily be reused. Creating a new test with slightly different inputs from an existing test should need very little effort. Manual tests can also be reused, but it is not as beneficial as in automatic testing since every manual test spends resources of a tester. Therefore, automatic testing makes it feasible to run many more variations of the same test.

Capability to perform certain tests: Some tests cannot be performed manually or they would be extremely difficult or economically infeasible to perform. Stress testing is usually easier to implement automatically than manually. For instance, testing a system with a large number of test users may be impossible to arrange. Another example is verifying events of a GUI that do not produce any immediate output that could be manually verified.

2.6 Automatic test systems

Automatic test equipment (ATE) is a machine that performs tests on a device or a system referred to as a *device under test (DUT)*, *unit under test (UUT)*, or *system under test (SUT)*. An ATE uses automation to rapidly perform tests that measure and evaluate the UUT. Complexity of ATEs ranges from simple computer controlled multimeters to complex systems that have several test mechanisms that automatically run high-level electronic diagnostics. ATEs are mostly used in manufacturing to confirm whether a manufactured unit works and to find possible defects. Automatic testing saves on manufacturing costs and mitigates the possibility that a faulty device enters the market. [10]

Automatic test system (ATS) is "a system that includes the automatic test equipment (ATE) and all support equipment, support software, test programs, and interface adapters" [3]. Based on these definitions it seems that an ATS is regarded as a specific test system designed and deployed to test a specific unit or units. An ATE, on the other hand, is an equipment which is designed for a specific purpose by a test equipment manufacturer, but it can be utilized for testing a greater variety of units. In other words, support equipment and test programs are required in addition to an ATE to actually perform testing on a specific unit.

ATE/ATS systems are widely used in the industry. Examples include testing consumer electronics devices, automotive electronic control units (ECU), life critical medical devices, wireless communication products, semiconductor components ranging from discrete components to various integrated circuits [21], and systems used in military and aerospace industries. [20]; [19]; [10]

3. TESTING FRAMEWORK CONCEPT

In this chapter a concept of a testing framework is presented. Key ideas of the concept are based on the requirements of performing durability and performance tests on rock drills. The most fundamental idea behind the concept is the usage of a testing methodology called *signal based testing*, which is described in Section 3.1. Test cases are defined in *test recipes*, which are discussed in Section 3.2. Test recipes are developed by using a graphical editor, whose functionalities are outlined in Section 3.3. Test recipes are executed in the ATS by a component called a *test engine*, described in Section 3.4. Section 3.5 discusses the need for data logging and how it can be implemented in a testing framework complying with the concept. An overview of the concept and how it can be utilized in implementing an ATS is illustrated in Section 3.6. Finally, Section 3.7 discusses applicability of the concept in various testing scenarios.

3.1 Signal based testing

Signal based testing is a testing methodology in which a system is tested through a signal interface. The system under test is regarded as a “box” which is stimulated with input signals and as a result of its operation the system produces output signals. The operation of the SUT can be verified to be correct by observing its outputs. Signal based testing can be performed by using either the black-box or gray-box testing strategy. [14]

Signal based testing relies on a generic interface to a SUT. Once the facilities for communicating with the system through the signal interface have been built, extending the interface should be relatively easy. When new inputs or outputs are required, they can simply be added to the signal interface and then used from the tests. From the tests’ point of view, no changes to the protocol between the test system and the SUT are required. In other words, the tests are decoupled from the internal implementation of the system, which is a major advantage in signal based testing. Another advantage is that the signal abstraction scales up well from small units to larger constructs; that is, there is no conceptual difference between unit, integration, and system testing. [14]; [11]

Definition of input signal types and verification criteria for expected results of output signals are discussed in Subsections 3.1.1 and 3.1.2, respectively. An alternative division to control and measurement signals is described in Subsection 3.1.3. Finally, the file format of the signal interface is described in Subsection 3.1.4.

3.1.1 Input signal types

Input signals have a type which defines the shape of the signal. Signal types have one or more configurable parameters. The simplest signal type is a constant function for which only a constant value must be defined. In some applications all use cases are covered with this signal type. However, some applications require more complex signal types such as an electrical AC (alternating current) signal, which has amplitude, frequency, and DC (direct current) offset as configurable parameters.

In addition to different signal shapes, the signal interface may need to support different data types. Both floating point values and integer values have advantages and disadvantages, and it may thus be beneficial to provide both. Moreover, Boolean values can be used for binary signals. Obviously, all signal shapes cannot be supported for all data types.

If signals are represented in the signal interface as floating point values, but the implementation behind the interface stores the values as integers, it has to be thought out whether rounding to integers may cause problems and how to avoid them. One means is to prevent test designers in the first place from specifying values that would be rounded. Alternatively, the issue might be just ignored if the provided precision is sufficient and no one will ever use values of greater precision.

3.1.2 Verification of output signals

Expected results of output signals are specified by means of verification criteria. The types of criteria that can be set on a signal depends on the signal's data type. Some possible criteria are listed in Table 3.1.

Table 3.1. *Typical verification criteria for expected results of output signals of different data types. [11]*

	Types of verification criteria			
Data type	Value	Equality	Range	Gradient
Floating point			X	X
Integer	X	X	X	X
Boolean	X	X		

The "Value" criterion compares an output signal to a specified value. If they are equal, the output of the system is regarded as valid. The "Equality" criterion compares two or more signals with each other. These checks can be performed on integers and Booleans only, because floating point values cannot be compared for equality. The "Range" criterion checks whether the output signal is within a specified range whereas the "Gradient" criterion checks whether the slope of the output signal is within a specified range. These

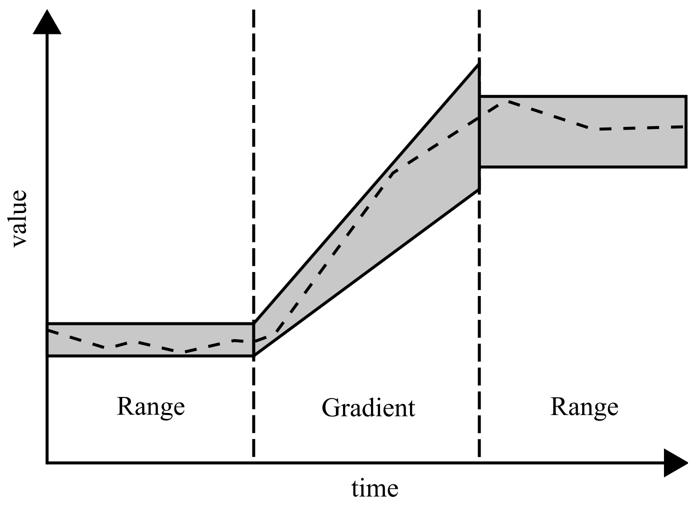


Figure 3.1. *An illustration of the "Range" and "Gradient" verification criteria. An output signal is shown as a dashed line. The grey areas are considered containing valid values for the output signal. The figure has been modified from the figure in source [11].*

criteria are illustrated in Figure 3.1. They can be used with floating point and integer signals but not with Boolean signals because it is not sensible to define a range for a variable that can have only two different values. [11]

3.1.3 Controls and measurements

In this subsection an alternative means for defining the signal interface is considered for testing some control systems. Control systems typically utilize measurements as feedbacks to controllers, which is called *closed-loop control* [5, pp. 1-4]. For most actuators there is a corresponding sensor whose data is used to control the actuator. The system might also have sensors for measuring quantities that are not directly controlled by any actuator, such as temperature. In addition to the division to input and output signals, the signal interface could alternatively be defined in terms of *controls* and *measurements*.

From a test designer's point of view, there is no need to separate the definition of a control function and the expected output to separate signals. It is also good practice to define the signal interface in a level which does not unnecessarily expose details of the implementation of the SUT. For instance, if percussion pressure is to be controlled and monitored, it can be represented as one control signal named "Percussion pressure" instead of having an input signal for the pump controlling the percussion pressure and an output signal for the sensor measuring it. Both input and output are related to the same quantity, so they can have the same name and data type.

Measurement signals are conceptually equivalent to output signals. The difference in the terminology, however, implies that all outputs of the SUT are measurements and not, for example, control signals to another system.

3.1.4 Signal definition file

The signal interface must be defined in a particular format defined by the implemented testing framework. It is useful to use a human-readable format so that the interface can easily be modified with a text editor without special tools. Then, the file also serves as a documentation of the signal interface for test designers. One good choice for the file format is XML (Extensible Markup Language) because it is widely used and there are a large number of XML parsers available for different programming languages.

An example of a signal definition file is shown in Figure 3.2. The XML consists of a top-level element `signals` that contains `signal` child elements. The signal interface has a version tag that is indicated in the `version` attribute of the `signals` element. `signal` elements have several attributes. The `signalName` attribute defines a unique identifier by which the signal is referred to by the testing framework. The `kind` attribute can be either “control” or “measurement”. Several attributes are related to how signals are represented when displayed in a GUI. `displayName` is the displayed name of the signal. `displayDataType` is the data type in which a user may specify values for the signal. This is not necessarily the same as the data type of the signal. The displayed unit of the signal is defined by the `unit` attribute. `scaleFactor` defines the scale factor that is used to convert a raw signal value to a displayed value. In this example implementation, all signals are of the same integer data type for simplicity; therefore, the data type is not explicitly specified in the signal definitions. `limitSignalTable` is related to defining verification criteria for the signal and will be discussed in more detail in Chapter 6.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<signals version="1">
   <!-- digital controls -->
   <signal signalName="LOC_DTB_WaterFlushingAutoTarget"
      kind="control" displayDataType="bool" unit=""
      scaleFactor="1.0" limitSignalTable=""
      displayName="Water flushing control"/>
   <!-- analog controls -->
   <signal signalName="LOC_DTB_PercussionAutoTarget_01bar"
      kind="control" displayDataType="double" unit="bar"
      scaleFactor="0.1"
      limitSignalTable="PAR_DTB_PercussionPressureLimits_01bar_1x6"
      displayName="Percussion pressure"/>
   <!-- measurements -->
   <signal signalName="LOC_DTB_TankOilTemperatureSensor_01C"
      kind="measurement" displayDataType="double" unit="C"
      scaleFactor="0.1"
      limitSignalTable="PAR_DTB_TankOilTemperatureLimits_01C_1x6"
      displayName="Tank oil temperature"/>
</signals>
```

Figure 3.2. *An example of a signal definition file.*

3.2 Test recipes

Automatic software testing usually means creating tests which are written in some programming language. The language is usually a general-purpose programming language and typically the same that was used to implement the software. This is an efficient way of performing automatic software testing since the tests can directly access the APIs (Application Programming Interface) of the software and no implementation of adapter layers is needed. Writing tests requires software expertise, but it is done by software specialists only, so this is not an issue. Automatic testing of devices or hardware components may, however, be performed by people that are not experts in programming. Therefore, automatic tests must be created by other means.

ATSs are computers, which require formal and unambiguous instructions on how to run a task. Although GUIs can be provided to the test designer, the test instructions must be stored in a format, which constrains how versatile and flexible the instructions can be. The format can be a programming language, a markup language with a certain schema, or some kind of a data structure as a binary representation.

A *test recipe* is here defined as a set of test instructions that a test designer creates and an ATS executes. The same term is also used to refer to the format in which the test instructions are represented. The test recipe format was designed based on the requirements in the customer project. The format can, however, be extended to support more advanced features.

Test recipes consist of *test phases* and *test loops*, which are described in Subsections 3.2.1 and 3.2.2, respectively. An example of a test recipe's structure is illustrated in Figure 3.3. Test recipes contain exactly one top-level item, which is a test loop. This loop may contain an arbitrary number of test phases and loops in a tree-like structure. Test recipes are stored in XML files that follow a specific XML schema, which is described in Subsection 3.2.3. Compatibility of test recipes with different versions of a signal interface is discussed in Subsection 3.2.4.

3.2.1 Test phases

A test phase is a test recipe item that contains test instructions for a particular test step. A test phase ends when an *end condition* specified for the test phase becomes true. In a versatile testing framework, the end condition could be any kind of Boolean expression that is dependent on the values of some signals and elapsed time. However, these kinds of complex end conditions are not discussed further; instead, only a simple end condition is discussed as an example.

The test designer specifies the duration of each test phase. When run successfully, a test phase ends when it has been run for the specified duration. It may, however, end prematurely if the verification of an output signal fails. This ends the test run altogether,

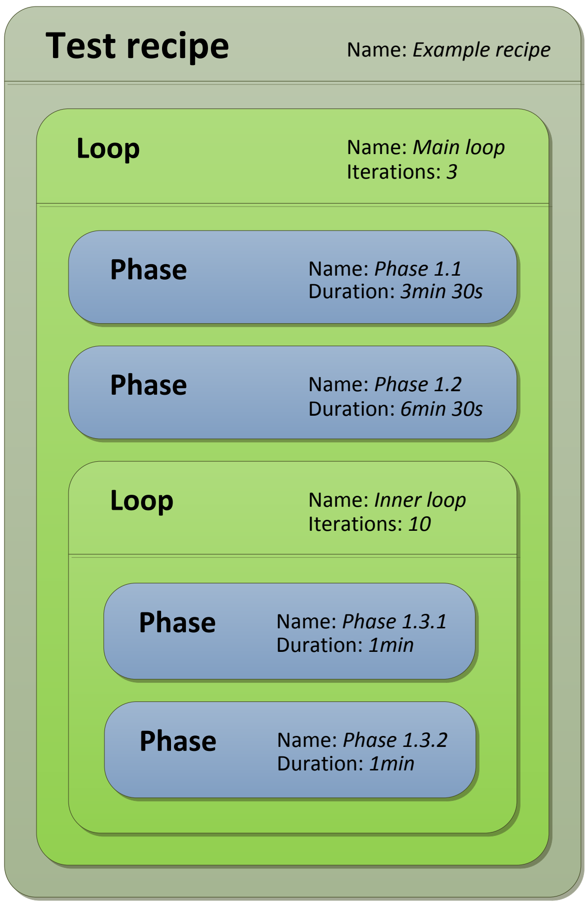


Figure 3.3. *An example of a test recipe's structure.*

and the test is regarded as failed.

A test phase includes definitions of inputs and expected outputs for that part of the test. Inputs and outputs are defined in terms of signals. In the described test recipe format, the signal interface is of the type described in Subsection 3.1.3. Control signals can only be constant values, and “Range” is the only supported type of a verification criterion. Control signals can be of a floating point or Boolean type. Measurement signals can be of a floating point type only.

Measurement signals typically contain some noise and temporary errors. Therefore, the noise needs to be filtered out by the test system to prevent false test failures due to a failed verification of a measurement signal. The filter could be defined separately for each test phase or test recipe, or the test system could have a global parameter which applies to all tests. It was decided in the customer project that a global parameter is sufficient for the time being.

3.2.2 Test loops

A test loop is a test recipe item that contains test phases and inner loops, which are run in a specified order. A test loop is run until an end condition specified for the test loop becomes true. The purpose of the test loops is to avoid unnecessary duplication of identical test steps. The test designer may want to, for example, loop certain test phases ten times, which would be a tedious and laborious task to perform without this feature. There is no imposed limit to the depth of the test loop structure.

In a versatile testing framework, an end condition of a test loop could be any kind of Boolean expression that is dependent on the values of some signals and elapsed time or loop iterations. However, complex end conditions are not discussed further here. The simplest solution for defining an end condition is to specify the number of times a test loop is run. Alternatively, the end condition could be defined in terms of elapsed time instead of iterations. However, this introduces a possible problem that needs to be taken into account. If the test designer specifies a test loop duration which is not a multiple of the duration of one iteration, the last iteration will be incomplete. This implies that the execution of a test phase might be stopped before the test phase is completed. This may be an issue especially if non-constant signal types are used in which case the input signals' phases, when the loop ends, are not clearly known at the time of implementing the test recipe. Moreover, the response might be slow for some control signals in which case it may have not reached the expected value when the test phase is changed. This may occur with constant function input signals as well.

3.2.3 Test recipe definition file

Test recipes are XML files that consist of a single top-level `recipe` element. An example of the element is depicted in Figure 3.4. The `signalDefinitionsVersion` attribute defines version of the signal interface. The test sequence is defined in a single `testLoop` child element.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<recipe name="Example test recipe" author="Tomi Kurko"
      signalDefinitionsVersion="1">
   <description>
      Description of the purpose and content of the test recipe
   </description>
   <testLoop/> <!-- Not shown here -->
</recipe>
```

Figure 3.4. *A test recipe XML file. Test recipes can have only one `testLoop` child element. The format of `testLoop` elements is shown in Figure 3.5.*

The test loop XML element is depicted in Figure 3.5. `testLoop` has attributes `name`

```
<testLoop name="Test loop" iterations="10">
   <description>Main loop of the test</description>
   <testLoop/> <!-- Not shown here -->
   <testPhase/> <!-- Not shown here -->
   <!-- An arbitrary number of test loops and phases can be included -->
</testLoop>
```

Figure 3.5. *A test loop XML element. The format of* `testPhase` *elements is shown in Figure 3.6.*

for the test loop's name and `iterations` for the number of iterations of the loop. The name need not be unique, although it is recommended for clarity. `testloop` can have an arbitrary number of `testLoop` and `testPhase` child elements.

The test phase XML element is depicted in Figure 3.6. Similarly to `testLoop` the attribute `name` need not be unique. `duration` defines duration of the test phase in seconds. `alarmEnableDelay` defines a delay in seconds before verification of signals is started. This is to prevent false test failures in the beginning of a test phase when the controls are still changing.

```
<testPhase name="Phase 1" duration="100" alarmEnableDelay="10">
   <description>Description of the phase</description>
   <control signalName="BOOL_CTRL_SIG">
      <setpoint value="true"/>
   </control>
   <control signalName="DOUBLE_CTRL_SIG">
      <setpoint value="30.00"/>
      <alarmLimits>
         <lowerLimit value="10.00" enabled="true"/>
         <upperLimit value="50.00" enabled="true"/>
      </alarmLimits>
      <warningLimits>
         <lowerLimit value="20.00" enabled="true"/>
         <upperLimit value="40.00" enabled="true"/>
      </warningLimits>
   </control>
   <measurement signalName="DOUBLE_MEAS_SIG">
      <alarmLimits>
         <lowerLimit value="0.00" enabled="false"/>
         <upperLimit value="95.00" enabled="true"/>
      </alarmLimits>
      <warningLimits>
         <lowerLimit value="0.00" enabled="false"/>
         <upperLimit value="85.00" enabled="true"/>
      </warningLimits>
   </measurement>
   <!-- An arbitrary number of signals can be included -->
</testPhase>
```

Figure 3.6. *A test phase XML element.*

The `testPhase` element can have an arbitrary number of `control` and `measurement` child elements. Both elements have the `signalName` attribute, which is the unique identifier of the signal. Verification criteria for the signal are expressed in terms of *alarm limits* and *warning limits*. These are "Range" type of verification criteria. Exceeding alarm limits causes the test to fail, but exceeding warning limits causes only a warning to be reported in the test results. `alarmLimits` and `warningLimits` elements have `lowerLimit` and `upperLimit` child elements. Both of them have a `value` attribute for specifying the limit value and an `enabled` attribute for specifying if the limit is enabled. `control` elements also have a `setpoint` child element. It has a `value` attribute which defines the value of a constant function.

3.2.4 Signal interface version compatibility

Test recipes are dependent on the signal interface used at the time of writing the test recipe. The signal interface may evolve and become incompatible with some old test recipes if backward compatibility is not maintained. Backward compatibility can be preserved if no signal definition is ever modified or removed but only new signals are added. Furthermore, the test system must not require that all signals have been configured in test recipes; otherwise, backward compatibility cannot be preserved in signal additions.

The test system must ensure that incompatible test recipes cannot be run. Therefore, the signal interface version is included in both the signal interface definition file and the test recipe definition file to be able to check for compatibility. The signal interface designer should update the version each time they make a change that makes the signal interface backward incompatible. Forward incompatibility may become an issue when the same test recipes are used in several test systems with different signal interface versions. In that case, a test recipe may use signals that do not exist in some test system using an older version of the signal interface. Therefore, a better solution would be to have three version numbers in the version tag in the format `x.y.z`. The major version number `x` would be increased when the changes break backward compatibility, and the minor version number `y` would be increased when only forward compatibility is lost. The third version number `z` could be used to indicate changes in the documentation of the signal interface, such as display names, that do not change functionality.

Regardless of the version tags it may occur that a test recipe is incompatible with the signal interface in use. This can be recognized by checking that all signals referenced by the test recipe exist in the signal interface and that the signal types match.

3.3 Graphical editor for test recipe development

Test recipes can be created and edited with a simple text editor. Since test recipes are XML documents, they can be edited with any textual or graphical XML editors, which can make editing somewhat faster and more convenient. However, even with graphical XML editors the process is tedious and requires knowledge of XML and the test recipe format. Therefore, it is recommended to provide a graphical test recipe editor tool for test designers so that they can do their work efficiently and focus on their expertise.

A graphical test recipe editor shall allow a test designer to create new test recipes and edit existing recipes. Basic file operations like New, Open, Save, and Save As shall be provided as in any program that is used for editing any kind of document. The editor may support editing multiple or only one test recipe at a time.

Basic operations required for defining a test sequence are addition and removal of test phases and loops. It is more convenient for the user if also the order of test items can be changed. All test items require some kind of an end condition that shall be editable. If the end conditions are based on time, durations of all test items and the overall duration of a test recipe shall be shown. Test items may also have additional textual information such as a name and description. Test recipes have at minimum a file name but may also have some other attributes such as a name, author, and description.

Some possible features are discussed in the following subsections. Subsection 3.3.1 discusses alternatives on defining signal configurations in test phases. Reusability of test sequences could be improved by supporting import and export of test sequences, which is discussed in Subsection 3.3.2. Resolving of incompatibility issues when updating test recipes to match a new signal interface version is discussed in Subsection 3.3.3. A concrete implementation of a test recipe editor, developed in the customer project, is described later in Chapter 4.

3.3.1 Signal configuration

Signals shall be configurable for each test phase. The configuration of a signal includes a selection of a signal type and definitions of signal parameters. The editor may either have the whole signal interface as a fixed set of signals that need to be configured explicitly for each test phase, or it may allow omitting a signal configuration when it is desired to be the same as in the previous test phase. This may affect the test engine implementation as well depending on how the latter option is implemented.

A fixed set of signals is easier to implement because then the editor can show the same list of signals for all test phases, and no functionality is needed for adding or removing signals from a test phase. It is also clear for a test designer where a signal configuration for a test phase comes from. A drawback is that this method may involve considerable duplication if most of the signals have same configuration for many test phases. Should the test designer need to change some signal configuration, they may need to do the same change for many test phases.

If each test phase has its own signal list, duplication can be avoided, but it comes with the cost of some complexity. If the user interface shows only the signals whose configuration changes in a test phase, it may be difficult for the user to see the overall signal configurations of the test phase. The user may need to browse through previous test phases to determine where a particular signal has been configured previously. This can be avoided if the user interface shows the effective signal configurations for the selected test phase and differentiates between those signals that are explicitly configured and those that are inherited from a previous test phase.

The decision between the two described options may also affect the test engine implementation. In the latter case the test engine must not assume that all signals have been configured for all test phases. This implies that it must know how to handle missing signal configurations. One option is to continue using a previously defined configuration if such exists. However, what should occur if a signal was configured to have a linear increase from a value A to a value B? Continuing to increase with the same slope is likely not what is desired. Adding a step function from B to A before the linear function is generally not desired either. One solution could be to handle a missing signal configuration as a constant function whose value is the end value of the latest configured signal type. In the linear function case, this would be B. The test designer might, however, want to use a sine wave for the whole duration of the test case in which case the constant function approach is not ideal either since duplication of configuration will be needed. Therefore, there is no general rule that works in all cases. Moreover, the test engine needs to handle the case that a signal is not configured in the first test phase by either regarding this as an error or by using some sensible default configuration.

Based on the reasoning above it seems that the fixed signal list is the simplest approach but may involve some duplication. Duplication could be avoided by adding a means to define a signal configuration by referencing another signal configuration in some other test phase. Whether avoiding duplication is worth the added complexity or not, depends on the case in question and needs to be considered in each separate case.

3.3.2 Importing and exporting test sequences

Test recipes may have commonalities in the test sequences. A test designer may therefore wish to be able to reuse some parts of existing test recipes. There are two options on how to implement this functionality which are *importing by copy* and *importing by reference*.

Importing by copy means reusing a test sequence by copying it from a test recipe to another test recipe. The test sequence can be the whole test recipe or part of it. This method is useful when a test designer wants to reuse most of the test sequence as is but possibly wants to modify some parameters. One benefit of this method is that the functionality can be implemented in the test recipe editor and no support from the test engine is required.

Importing by reference means reusing a test sequence by having a link from the test recipe where the test sequence is desired to be included to the test recipe where the test sequence is defined. This implies that changes to the original test sequence affect all test recipes that link to it. In this case it may be easier to consider only importing whole test recipes. This method may be beneficial, for example, when some common test sequence needs to be run in several test recipes prior to running the actual test. However, an alternative solution for this particular use case could be to instruct the person performing testing to run a particular test recipe prior to starting testing, for example, a test recipe designed for warming the oils of a drill test bench. Implementing the import by reference functionality is more complex since support is needed for both the test recipe editor and the test recipe engine.

Exporting test sequences would be useful if also import by reference was implemented. A test designer might want to export part of a test recipe, which they would import to another test recipe. One way to implement the export functionality is that a new test recipe is created and the exported test sequence is added to it.

3.3.3 Updating test recipes against new signal interface version

As described in Subsection 3.2.4, test recipes need to be compatible with the signal interface that is in use. Incompatible test recipes can be updated by manually editing the files, but this may become cumbersome, especially if the signal interface is changed frequently. In this case it may be worthwhile to develop features for resolving incompatibility issues in the editor.

Problems that may be encountered during a test recipe update and possible actions for resolving them are listed in Table 3.2. Note that some actions for resolving a problem may actually raise a new problem of a different kind. The process is continued until all problems have been resolved.

3.4 Test engine

The component of the testing framework which executes the test recipes is called a *test engine*. Tasks of the test engine include keeping track of elapsed time, changing the current test step when the end condition is met, controlling input signals, and verifying output signals. A flow chart of the test engine's operation is depicted in Figure 3.7.

The test engine controls the SUT by stimulating it with input signals. While the test is running, it constantly monitors the system's outputs and verifies them against the defined criteria. Should a verification fail, the test engine stops the execution immediately and regards the test as failed. Depending on the application there might also be other reasons for the test execution to be aborted such as an emergency stop. The reason for a test failure is recorded in test results. On a successful test run, the test recipe is run until all test steps have been executed.

Table 3.2. *Possible problems and actions for resolving them when updating a test recipe against a new signal interface version.*

Problem	Action
Reference to an unknown signal.	Suggest removing the configuration or selecting a signal that corresponds to the old signal.
Mismatching data type (input signal).	Suggest automatic fixing if the configured signal type is supported with the new data type; otherwise, suggest selecting another signal type and configuring its parameters. In both cases removal of the configuration shall also be possible.
Mismatching data type (output signal).	Suggest automatic fixing if the configured verification criterion is supported with the new data type; otherwise, suggest selecting another criterion type and configuring its parameters. In both cases removal of the configuration shall also be possible.
Missing signal configuration (if a fixed signal list is used).	Warn about a missing configuration, and add a default one.

The test engine communicates with the SUT by means of a *data access interface*, which is an interface for accessing data of the signals defined in a signal interface. It provides a generic means for the test engine to read and write signals and makes it immune to changes in the signal interface. If the SUT implements the data access interface used by the test engine, the test engine can communicate with the SUT directly. Otherwise, a *SUT data access (DA) adapter* is needed. This adapter adapts the SUT's interface to the data access interface. In practice, the adapter may be a software module or a piece of hardware that interacts with the SUT depending on the interface of the SUT. Changes in the signal interface reflect to the implementation of the adapter.

The choice of a data access interface depends on the system that is going to be tested. If the SUT already supports some data access interface, that interface can be used in the test engine as well, eliminating the need for an adapter. This makes the ATS generic in the sense that it can be used to test all systems that have the same data access interface and not only one specific system.

One example of a standard data access interface is OPC DA (Open Platform Communications, Data Access). The OPC Foundation has created a number of software interfaces that aim to standardize the information flow from the process level to the management level in industrial automation applications. Data Access is the first and most successful Classic OPC Standard; it was implemented in 99% of the products using OPC technology in 2009. It enables reading, writing, and monitoring of variables containing current

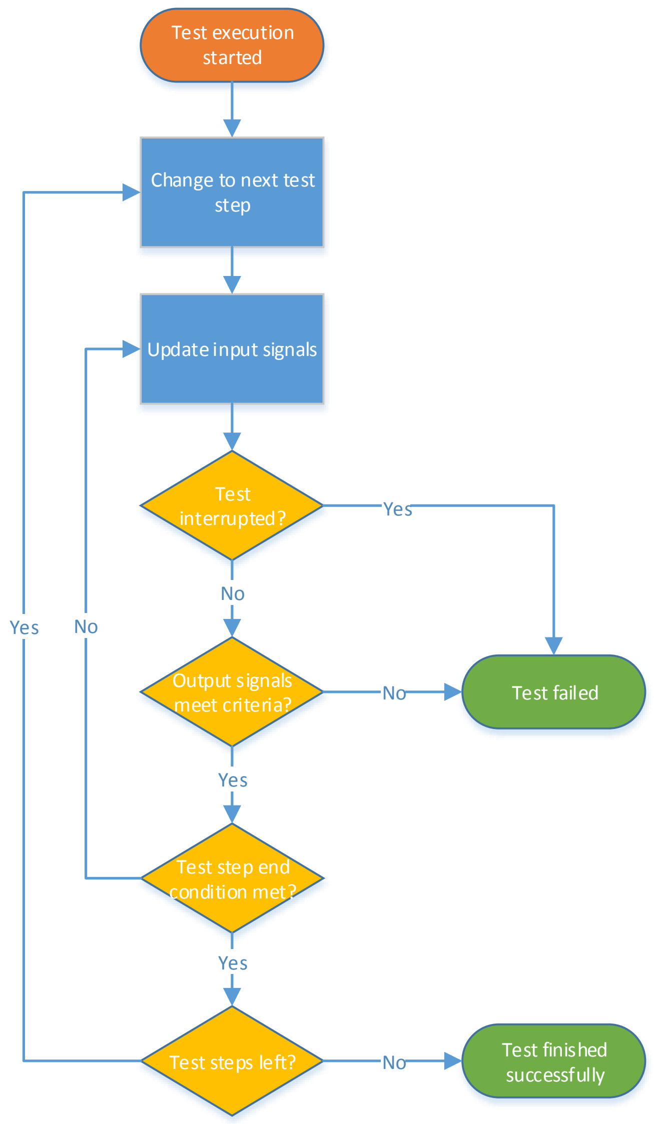


Figure 3.7. *Operation of the test engine.*

process data. Manufacturers of Soft PLCs (Programmable Logic Controller) and most of the HMI (Human-Machine Interface), SCADA (Supervisory Control and Data Acquisition), and DCS manufacturers in the field of PC-based automation technology offer OPC interfaces with their products. Classic OPC interfaces are based on the COM (Component Object Model) and DCOM (Distributed Component Object Model) technologies used in Microsoft Windows operating systems. Main disadvantages of Classic OPC are the dependency to the Windows platform and the DCOM issues, such as poor configurability, when using remote communication with OPC. The first version of the OPC DA specification was released in 1996. Classic OPC has since been superseded by OPC UA (Unified Architecture), which, in contrast to Classic OPC, is platform-independent. Some of the other benefits over Classic OPC include increased reliability in the communication between distributed systems, and an object-oriented, extensible model for all OPC data.

[16, pp. 1, 3-4, 8-9]; [11]

Real-time requirements of the test engine depend on the system that is going to be tested. If the SUT is not a hard real-time system or its behaviour need not be verified very accurately with respect to time, the test engine could be implemented in a soft real-time system. Failure to operate exactly according to the defined timings in a test recipe would then result in only minor deviations from the test specification. For instance, in durability testing, where test runs can last tens of hours, an error of the order of a second in total time is completely meaningless. Accuracy in changing a test step is likely to be unimportant as well if test phases are at least tens of seconds long.

3.5 Data logging

In some cases, verifying the operation of the SUT completely automatically may not be feasible. For instance, in testing of mechanical systems, functionality of the SUT might be automatically verified, but performance may be easier to analyze manually. It is possible to implement automatic performance tests to some extent, but it may be necessary to leave some of the verification for a test engineer to perform manually. Manual inspection of the operation of the SUT requires extensive amount of data, which can be logged during an automatic execution of a test. The test engineer may use mathematics software like MATLAB to analyze the data.

Signal based testing brings the benefit that data logging can be implemented easily due to the generic signal and data access interfaces. The data logging component may simply query the values of all signals or some specified selection of the signal set at regular intervals and store the data to a file or a database. Data logging can be implemented as part of the test engine or as a separate component.

Depending on the application there might be need for collecting data at several measuring rates. For instance, it may be required to get the most recent data at high measuring rate so that if the test fails, the reason for the failure can be identified accurately from the test data. On the other hand, due to practical difficulties in handling large amounts of data, it may be necessary but also sufficient to store most of the test data at a lower sampling rate. High sampling rate imposes hard real-time requirements for the measurement and data logging system.

3.6 Test system overview

The ideas presented in this chapter can be joined to form a concept-level architecture of a testing framework that can be used in automatic test systems. An overall visualization of the concept is illustrated in Figure 3.8.

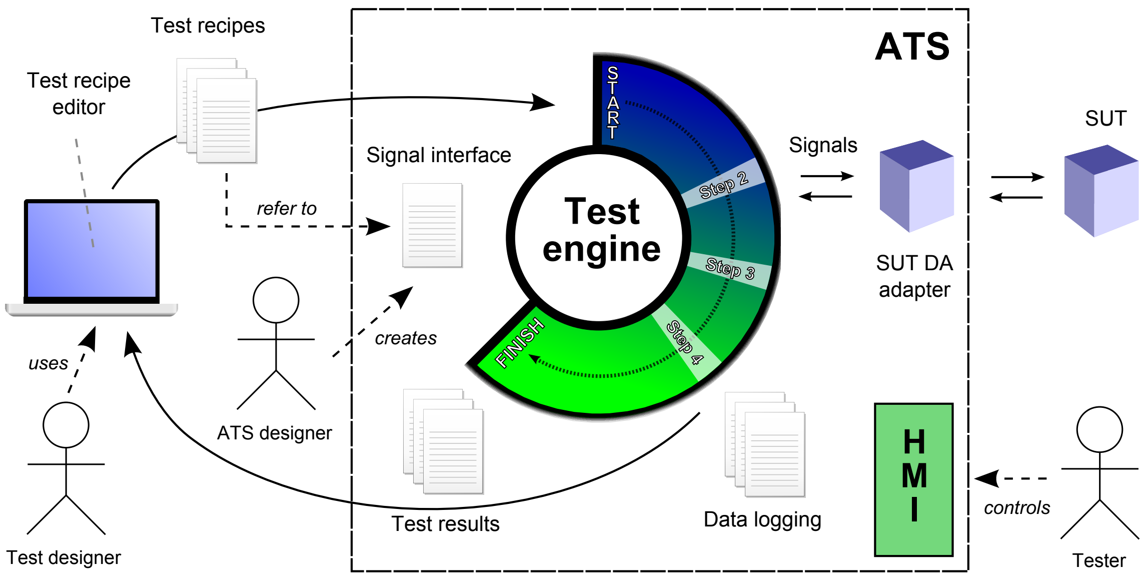


Figure 3.8. *A visualization of the testing framework concept.*

Before creating tests, the signal interface is defined. This is the task of the ATS designer. The interface needs to be rather stable; otherwise, tests may become obsolete and need updating if backward compatibility is not maintained. Test designers implement test recipes by using the graphical test recipe editor on their desktop computers. When one or more test recipes have been created, they can be loaded in the ATS. The tester starts a test by using the HMI of the ATS, and the test engine will start to execute the loaded test recipe. After the test has been finished, the tester may copy the test results and test data to their desktop computer and perform further analysis.

Details of the test system architecture may vary depending on the requirements of the application. For instance, the stimulation of the SUT with input signals may involve running PID (Proportional-Integral-Derivative) controllers on a hard real-time system. If the test execution, however, is not time-critical, the test engine might be divided into higher and lower level components that are run on different hardware. Requirements for data logging also affect which hardware is needed and what kind of software components are needed to implement the test engine and data logging functionalities.

3.7 Applicability to various testing scenarios

The testing framework concept described in this chapter is very generic and could theoretically be applied to many testing scenarios. Main characteristics of the concept are summarized in the following list:

- Tests can be created and run by non-programmers. No programming skills are required for being able to use the graphical editor tool.

- Tests are decoupled from the SUT by means of a signal interface. The signal interface is the only interface between the tests and the SUT, and the interfaces of the SUT itself need not be exposed to the tests.
- The framework can be used for testing software, hardware, or whole systems. The signal abstraction is suitable for testing many kinds of systems.
- Tests are run in real-time; there is no control over execution of the SUT (if it is running software). The test engine is run independently of the SUT, which needs to be run by itself.
- Exact data synchronization is not possible. Since there is no control over execution of the SUT, input and output data cannot be synchronized.
- Parts of a test can be repeated multiple times. Test loops can be used to group test items which are run several times.
- Duration of a test is known beforehand. When using the described static end conditions of test items, all test runs of a test recipe are equal in terms of duration.
- Control flow of the tests is static; there is no dynamic behaviour at run-time. When using the described static end conditions of test items, all test runs of a test recipe are equal in terms of the test steps that are executed.
- With the addition of dynamic end conditions that are dependent on the values of signals at run-time, length of test items could be varied at run-time. This also implies that test steps could be skipped at run-time based on a conditional expression.

The list of characteristics above illustrates that there are advantages in the concept, but as such it has characteristics that limit its applicability to various testing scenarios.

The concept is obviously not intended for testing desktop PC or mobile software since there are better testing methodologies and tools for that. As for the software used in embedded systems, its business logic can, in most cases, be tested in the PC development environment without the target hardware. Therefore, the concept is not ideal for testing such software either. However, in embedded systems, there are also the hardware and the co-operation of software and hardware that affect how the system works. Testing the embedded system as a whole may be reasonable with this concept in some cases. System testing of an industrial controller code is an example where the concept can be useful. However, some more support from the test system like synchronization to the controller's control cycle might be needed [11].

Testing of DCS systems is a potential use case as well. No synchronization is needed when testing the system as a whole since the different modules may have different control cycles, and it is the overall response time that counts. Typically, the modules communicate via a fieldbus by using some kinds of signals, which makes signal based testing feasible. For testing DCS systems the concept scales up well from smaller systems to more complex ones.

The concept works best at similar testing scenarios to rock drill testing; that is, testing performance and reliability of mechanical, hydraulic, and electrical devices. Some improvements and extensions, however, might be needed to the test recipes to satisfy requirements of the particular application domain. These are discussed further in Section 7.2.

4. DEVELOPMENT OF A GRAPHICAL TEST RECIPE EDITOR

This chapter describes the implementation of a test recipe editor made in the customer project. The editor was designed for the purpose of creating test recipes for a drill test bench. However, usage of a signal interface allowed the implementation to be generic so that it does not have any application domain specific information. Possible features for a test recipe editor were outlined in Section 3.3, but not all of them were implemented in the customer project. Design of the implemented editor is discussed in Section 4.1. Technology choices and other details of the implementation are described in Section 4.2. The GUI of the editor and the implemented features are described and depicted in Section 4.3.

4.1 Design

The most important requirement that affected the editor's GUI design was that the editor needs to be usable in two different environments: an industrial PC with a touch screen, and desktop and laptop computers with mouse and keyboard. The industrial PC is used in the drill test bench, and it has a 12-inch touch screen whose resolution is 1024x768 pixels. When designing a GUI for a touch screen, the GUI elements need to be larger in size so that the user can easily press buttons and other controls with their finger and read the text on the screen. On the other hand, in desktop software the font and the controls can be smaller since pointing with a mouse is more precise. Both environments need to be supported, but the major use environment is desktop computers. Due to the development time resources being limited, it was decided that the user interface used on both of these environments would be mostly similar. The layout is identical in both platforms, but the widgets need to be slightly different to support the different input devices.

In Section 3.3 possible features of a test recipe editor were discussed. The implemented editor was designed to allow the editing of multiple test recipes at a time. Test recipes and the signal interface definition follow the file formats described in Subsections 3.1.4 and 3.2.3. The signal set is fixed, and all signals have to be configured separately in each test phase. Only constant control functions and constant limits for monitoring output signals are supported as per the test recipe format. Importing test recipe items from other recipes is currently not supported.

When new test recipes are created, they are created based on the signal interface definition. If an error occurs in reading the definition file, the editor cannot be used and only an error message is shown blocking all functions of the editor. Subsection 3.3.3 discussed possible incompatibility issues between a test recipe and the signal interface. These situations must be resolved manually by editing the incompatible test recipe manually, and the editor only shows an error dialog describing the encountered problem.

4.2 Implementation

The editor was implemented with Qt 4.7.1, which is a cross-platform application and user interface framework. Qt 4.7.1 was released in November 2010 [17], so it is already 3.5 years old at the time of writing (June 2014), and several new releases, including a new major release Qt 5, has been made after that. [28] The reason for choosing Qt 4.7.1 was that the customer's SICA SUP platform (Sandvik Intelligent Control Architecture, Supervisor) is based on it and the editor had to be binary compatible with it. The cross-platformness of Qt allows the software to be easily compiled for Windows and Linux operating systems. The C++ language and the Qt's model/view architecture with C++ widget classes were used in the implementation [27]. In contrast, Qt Quick contains the JavaScript based QML language used for creating GUIs. However, Qt Quick was not an option since it was officially released in Qt 4.7.2 [33].

Design of the GUI was guided by the requirement that the GUI needs to be usable on a touch screen. In the touch screen port, widgets from the SICA SUP platform were used, which show a virtual keyboard on the screen when an input field is touched. Another reason for using SICA widgets was the desire to have consistent look and feel of the user interface throughout Sandvik's software. In the desktop port, however, dependency to SICA was not desired since it was by no means necessary, and only Qt's widgets were used.

The user interfaces were designed with Qt Designer, which uses the *user interface definition (.ui)* file format. The XML format .ui files are converted to C++ code by the User Interface Compiler (uic), which reads a .ui file and creates a corresponding C++ header file. [30] The decision to use different widgets in different ports implied that separate .ui files had to be used for the ports, which increases development and maintenance effort. However, nearly all of the C++ code is the same in both ports; therefore, the effort is considerably smaller than in implementing two totally different user interfaces. Instead of using conditional compilation, which in many cases results in complicated code, the differences in the code were placed in separate source files, which are selected based on the platform by the build system.

XML reading and writing was implemented by using Qt's QXmlStreamReader and QXmlStreamWriter classes. The content of a test recipe XML is loaded to a data structure consisting of a set of classes defined in the `libTestRecipe` library, which is shared with the test engine implementation. This solution makes most of the code independent of the file format that is used. Writing the data model to an XML file was implemented by using the *Visitor pattern*. According to Gamma et al. a visitor represents an operation to be performed on the elements of an object structure. A major benefit in using the Visitor pattern is that it enables new operations to be defined without changing the classes of the elements on which they operate. [8, p. 331] In the case in question, the Visitor pattern makes the implementation more easily extensible to new file formats. It has also enabled the implementation of some functions to be simpler, such as checking the recipe for unconfigured signals.

The package diagram of the editor is depicted in Figure 4.1. The platform-specific code includes the .ui files and a little amount of C++ code for handling some minor differences between the ports. A main program is also included in the desktop port but not in the touch screen port, as it is provided by SICA. Most of the code is, however, shared between the ports as libraries or source files. The `libTestRecipe` library includes the classes that form the data structure of test recipes. `libTestRecipeEditor` implements all models of the editor in accordance with the Qt's model/view framework. The models work as adapters between the views and the actual data store. Since the view classes depend on the user interface definitions, which are platform-specific, they cannot be built into a library. Instead, they are provided as source files by the `TestRecipeEditor_platform` package. Utilities for reading the signal interface definition XML file and performing various file operations for the test recipes are included in the `libTestRecipeUtils` library. `libTimeUtils` provides some generic time related utilities, and `Common` provides some other commonly used utilities.

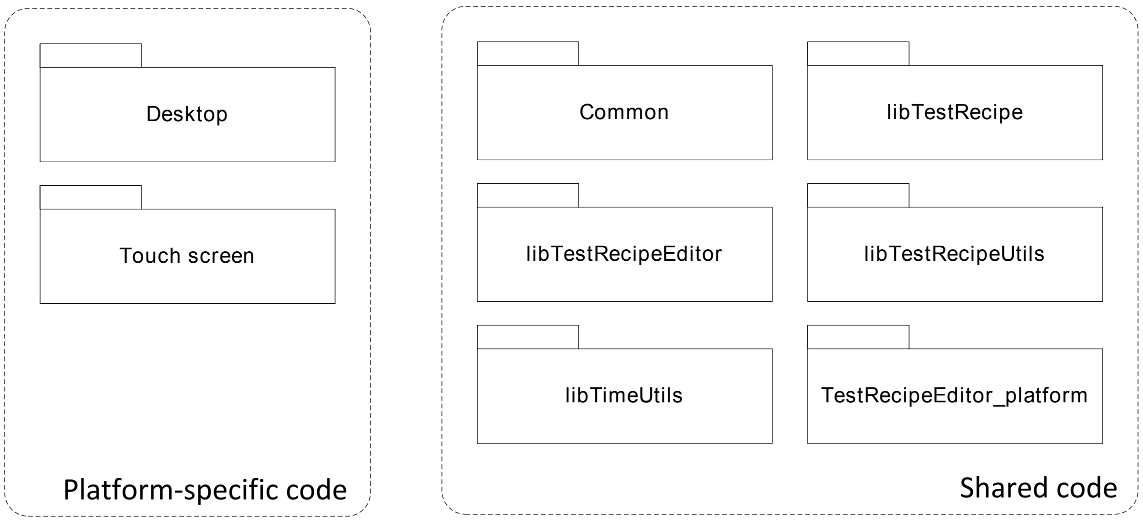


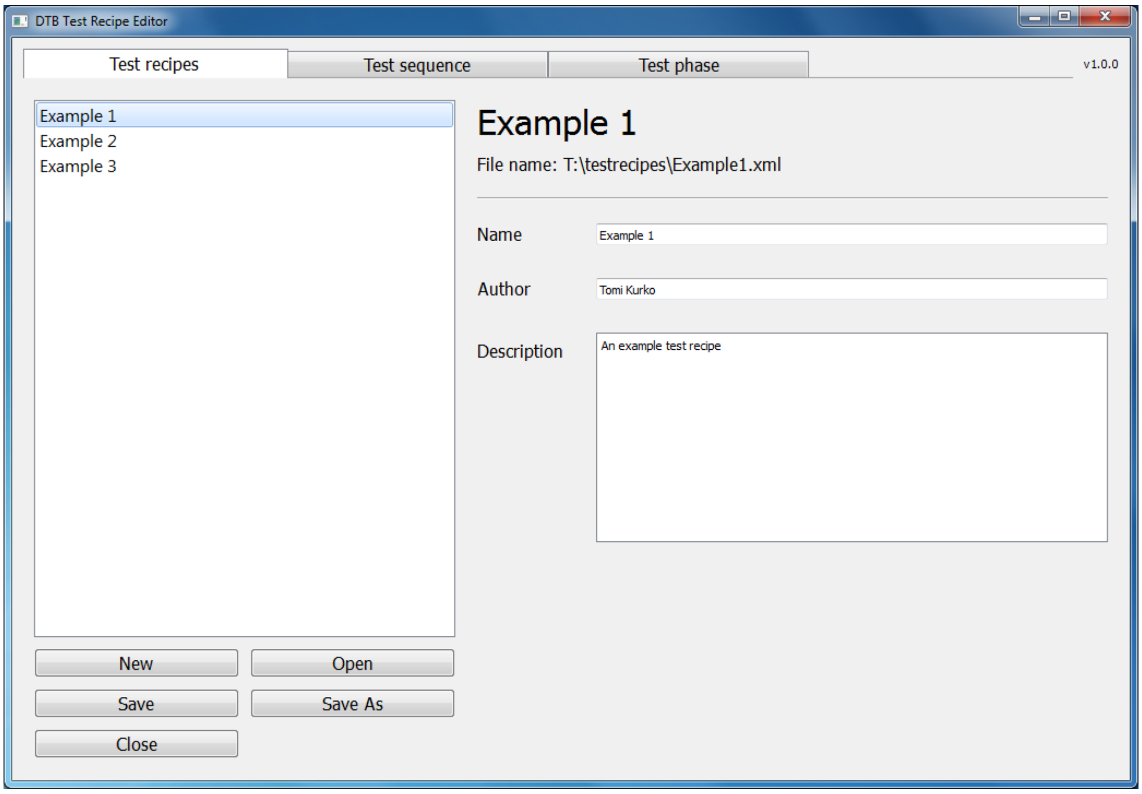
Figure 4.1. *The package diagram of the test recipe editor.*

4.3 User interface

The user interface is divided into three views, which are selectable from a tab bar topmost in the user interface: *the test recipe view*, *test sequence view*, and *test phase view*. The user selects the recipe to be edited in the test recipe view. When a recipe has been selected, the other tabs become available. In the test sequence view, the user may edit the sequence of test items, that is, test loops and phases. The signal configurations of test phases are edited in the test phase view. The test recipe, test sequence, and test phase views are described separately in more detail in Subsections 4.3.1, 4.3.2, and 4.3.3, respectively.

4.3.1 Test recipe view

A screenshot of the test recipe view is shown in Figure 4.2. In this view, open test recipe files are shown, and the user can create new recipes, open existing recipes, and save and close open recipes. The open recipes are shown by their name in a list view on the left side of the test recipe view. On the right side, information on the recipe, including its name, author, and description, is editable in input fields. The name and file name of a test recipe need not be the same, and the file name is shown with a full path separately.



Example 1

File name: T:\testrecipes\Example1.xml

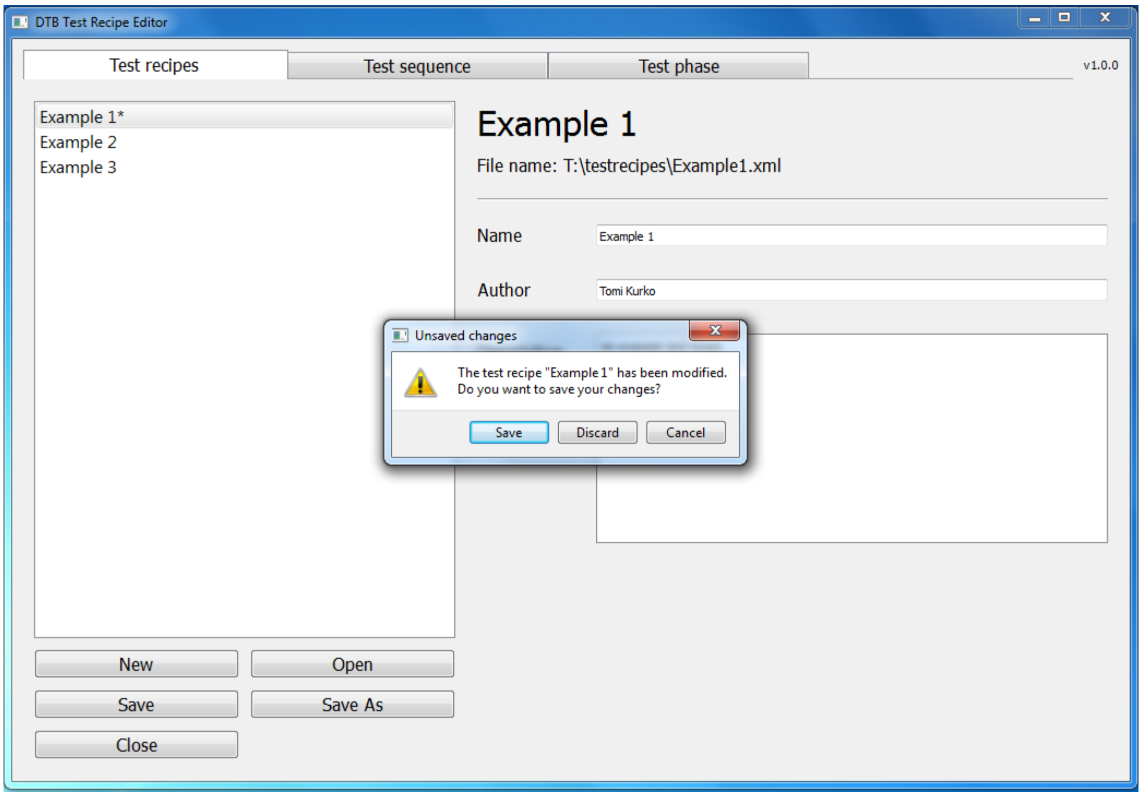
Name: Example 1

Author: Tomi Kurko

Description: An example test recipe

Figure 4.2. *The test recipe view.*

One important feature of any document editor is the indication of unsaved changes and asking the user a confirmation if they attempt to close a document that has unsaved changes. Unsaved changes are indicated in the list view by appending an asterisk to a test recipe's name. An asterisk is used in many programs in the same purpose, so it should be familiar to the user. Closing an unsaved test recipe is illustrated in Figure 4.3. When trying to exit the program, it tries to close the open test recipes one at a time in the order they are in the list. The confirmation dialog will be shown if there are unsaved changes. If the user presses the "Cancel" button, the program will not exit, and the rest of the test recipes will not be closed.



Example 1*
Example 2
Example 3

Example 1

File name: T:\testrecipes\Example1.xml

Name: Example 1

Author: Tomi Kurko

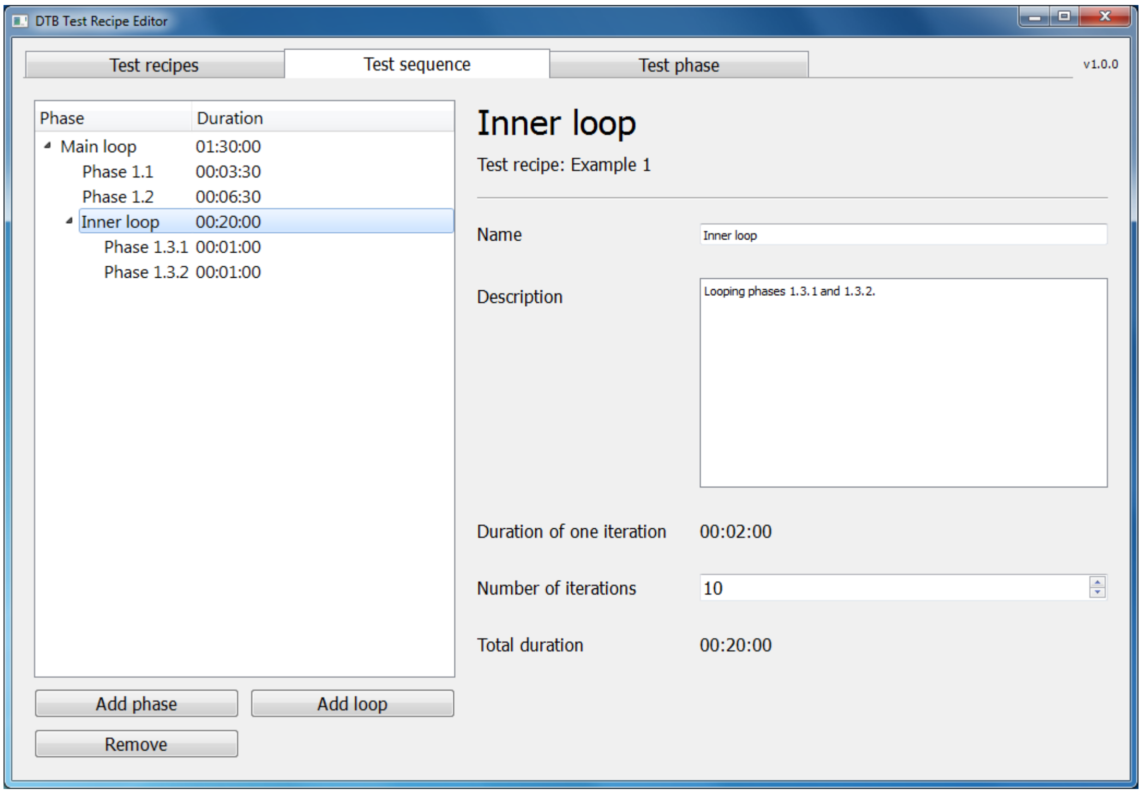
Unsaved changes

The test recipe "Example 1" has been modified.
Do you want to save your changes?

Figure 4.3. *A confirmation dialog is shown when the user tries to close a test recipe that has unsaved changes. Modified test recipes are indicated in the list view with asterisks.*

4.3.2 Test sequence view

Screenshots of the test sequence view when editing a test loop or test phase are shown in Figures 4.4 and 4.5, respectively. In both cases a tree view of the test sequence is shown on the left side of the view. It shows all test loops and phases of the test recipe and their durations in hours, minutes, and seconds. The user can add new loops and phases into a test loop by selecting the loop and pressing the buttons below the tree view. A selected item can be removed by pressing the "Remove" button. It can also be moved by dragging and dropping it either over a test loop's name, between two items, or at the end of a test loop. During the drag operation, indicators are shown which indicate where the item will be placed. Dropping an item over a test loop's name appends the item to the test loop. If


DTB Test Recipe Editor

Test recipes | Test sequence | Test phase — v1.0.0

Phase	Duration
Main loop	01:30:00
Phase 1.1	00:03:30
Phase 1.2	00:06:30
Inner loop	00:20:00
Phase 1.3.1	00:01:00
Phase 1.3.2	00:01:00

Add phase · Add loop · Remove

Inner loop

Test recipe: Example 1

Name	Inner loop
Description	Looping phases 1.3.1 and 1.3.2.
Duration of one iteration	00:02:00
Number of iterations	10
Total duration	00:20:00

Figure 4.4. *The test sequence view when editing a test loop.*

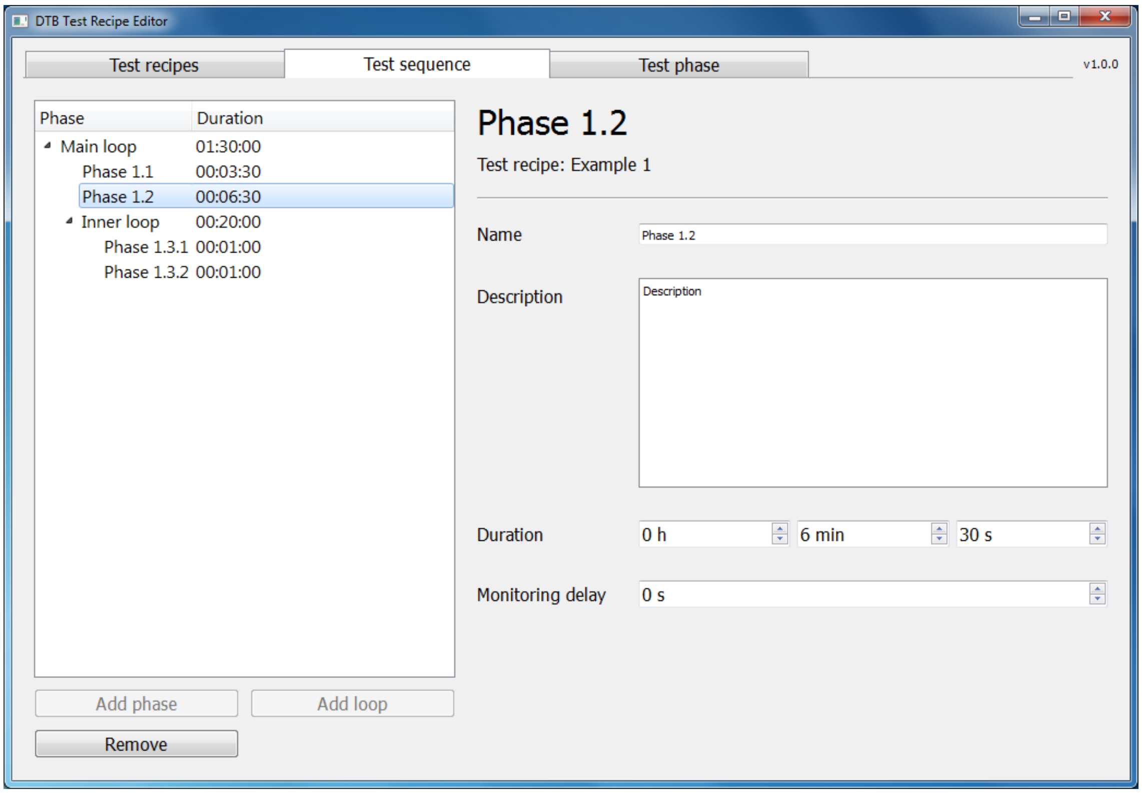
there are no items in the test sequence, for example, when a new recipe was created, only adding a loop is possible and other functions are disabled.

When a test loop has been selected from the tree view, the user may edit the name, description, and the number of iterations in the loop. The duration of one iteration is calculated by summing the durations of the child items of the loop. Total duration of the loop is also shown. Both durations are shown in hours, minutes, and seconds.

Similarly to test loops, the name and description of a test phase can be edited when it has been selected from the tree view. Additionally, the duration of the test phase can be edited by adjusting hours, minutes, and seconds separately, and the delay between starting the test phase and starting monitoring can be set in seconds.

4.3.3 Test phase view

In the test phase view, the signal configurations of a test phase are defined. A screenshot of the test phase view is shown in Figure 4.6. In the top-left corner, there are two dropdown lists for selecting which signal configurations to edit. The upper one opens a tree view similar to the one in the test sequence view from which the test phase to be edited can be selected. The lower dropdown list has two options: one for showing only control signals, and the other for showing only measurement signals. The signals are shown in a table view below the dropdown lists. The table view is described in more detail later. Contents



Phase	Duration
Main loop	01:30:00
Phase 1.1	00:03:30
Phase 1.2	00:06:30
Inner loop	00:20:00
Phase 1.3.1	00:01:00
Phase 1.3.2	00:01:00

Phase 1.2

Test recipe: Example 1

Name	Phase 1.2
Description	Description
Duration	0 h / 6 min / 30 s
Monitoring delay	0 s

Figure 4.5. *The test sequence view when editing a test phase.*

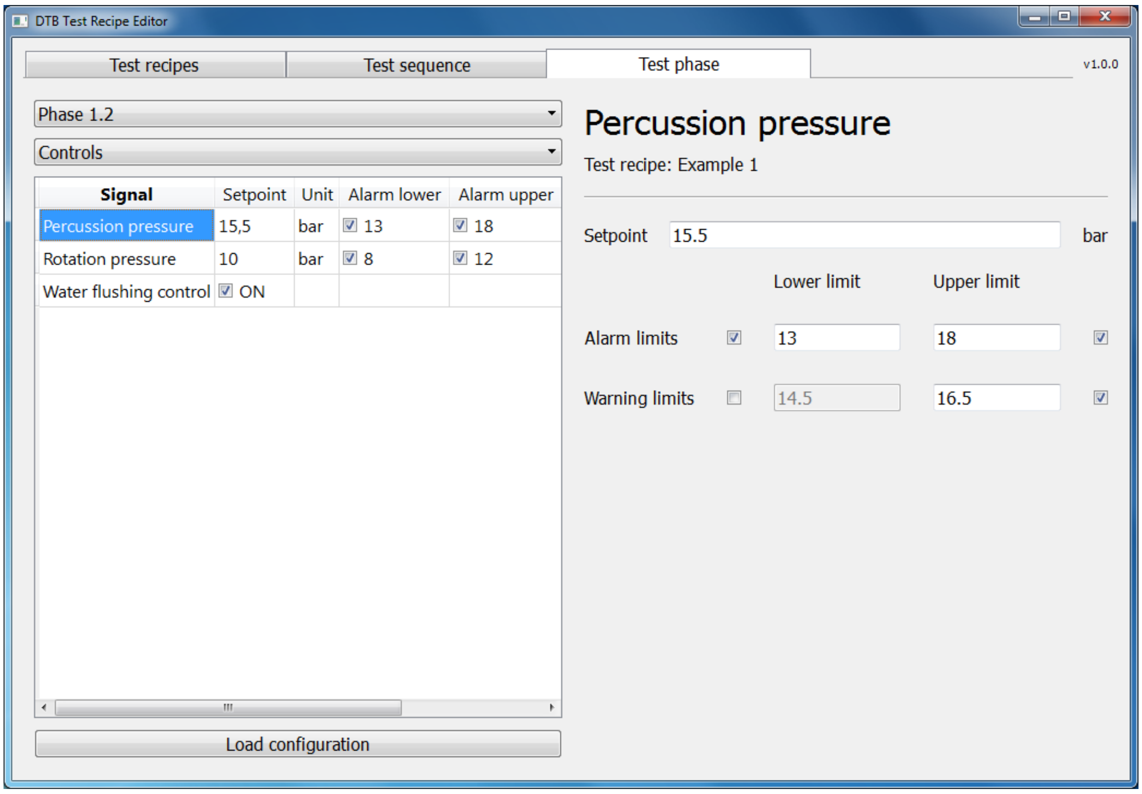
of the right side of the view depend on the signal type that is being configured.

Figure 4.6 shows how the view looks like when editing a floating point type control signal. The constant setpoint can be set in an input field which accepts real numbers. If the precision of a specified value is higher than what is supported by the signal interface, the value is rounded after focus leaves the input field. The unit of the signal is shown next to the input field. Warning and alarm limits, both including a lower and an upper limit, can be enabled or disabled separately via checkboxes. Each limit value is defined as an absolute value in the unit of the signal being edited. An option to define the limits as relative to the setpoint was discussed but was not implemented for the time being. Values of disabled limits are stored as well.

When configuring a digital, Boolean type control signal, the view is very simple as shown in Figure 4.7. There is only one check box, which is used to configure the constant setpoint, either on or off, of the signal. Boolean type signals cannot currently be monitored.

Configuring a floating point type measurement signal is shown in Figure 4.8. In this case the limits are configurable the same way as in the case of a floating point type control signal.

Originally, the left side of the test phase view was designed to have a list of signals without any information on the configurations. However, it was discussed with the customer that it would be preferable to have at least all control signal setpoints visible at the



Signal	Setpoint	Unit	Alarm lower	Alarm upper
Percussion pressure	15,5	bar	☑ 13	☑ 18
Rotation pressure	10	bar	☑ 8	☑ 12
Water flushing control	☑ ON			

Percussion pressure

Test recipe: Example 1

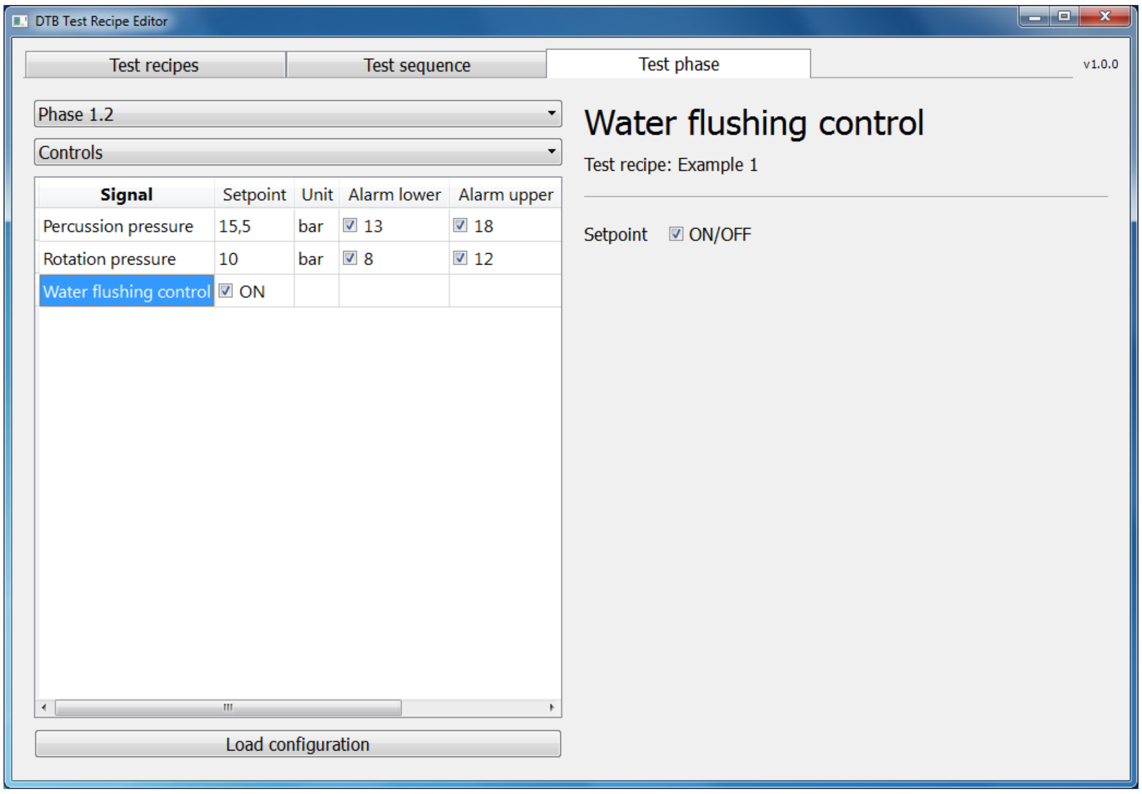
Setpoint 15.5 bar

		Lower limit	Upper limit	
Alarm limits	☑	13	18	☑
Warning limits	☐	14.5	16.5	☑

Figure 4.6. *The test phase view when editing a floating point type control signal.*

same time. In the list approach the user would need to go through all control signals by selecting them one at a time to see how a test phase is configured. Therefore, the list view was replaced with a table view. The width of the table view can be adjusted by using a slider next to the table view. The right side of the test phase view can also be hidden altogether as shown in Figure 4.9. Currently, the table view displays all the same information that is shown in the right side of the test phase view except the test recipe's name. It is also editable in a way similar to a spreadsheet program. Therefore, it may seem like there is little use for the original means of representing the data. However, if more signal types were added, it might be difficult to represent the data in a table form. Adding new columns that would be relevant only for some signal types might make the table confusing and difficult to edit if there are too many columns. Thus, the signal type specific views in the right side of the test phase view may be more useful at a later date.

Signal configurations can be loaded from another test phase to the currently selected one by pressing the "Load configuration" button. This shows a dialog shown in Figure 4.10. All signal configurations of the currently selected test phase are replaced with the ones from another test phase. This feature can be used, for example, when creating new test phases that would be mostly similar to an existing test phase.



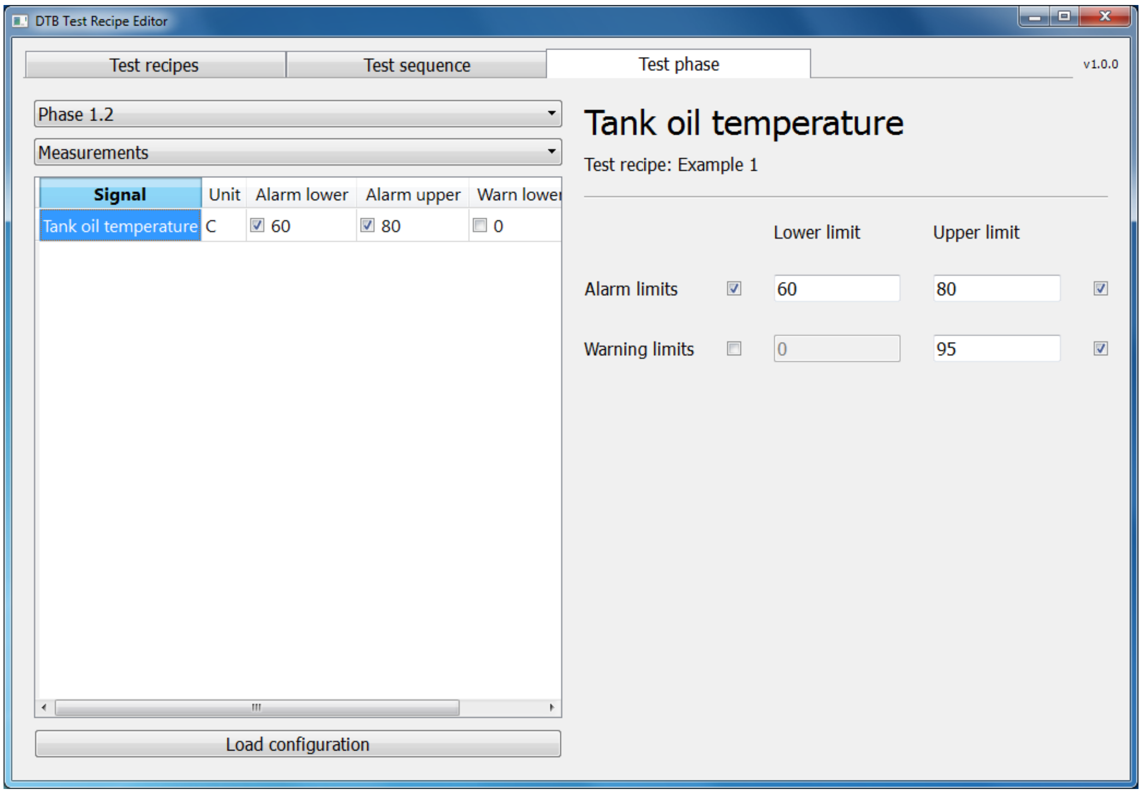
Signal	Setpoint	Unit	Alarm lower	Alarm upper
Percussion pressure	15,5	bar	☑ 13	☑ 18
Rotation pressure	10	bar	☑ 8	☑ 12
Water flushing control	☑ ON			

Water flushing control

Test recipe: Example 1

Setpoint ☑ ON/OFF

Figure 4.7. The test phase view when editing a Boolean type control signal.



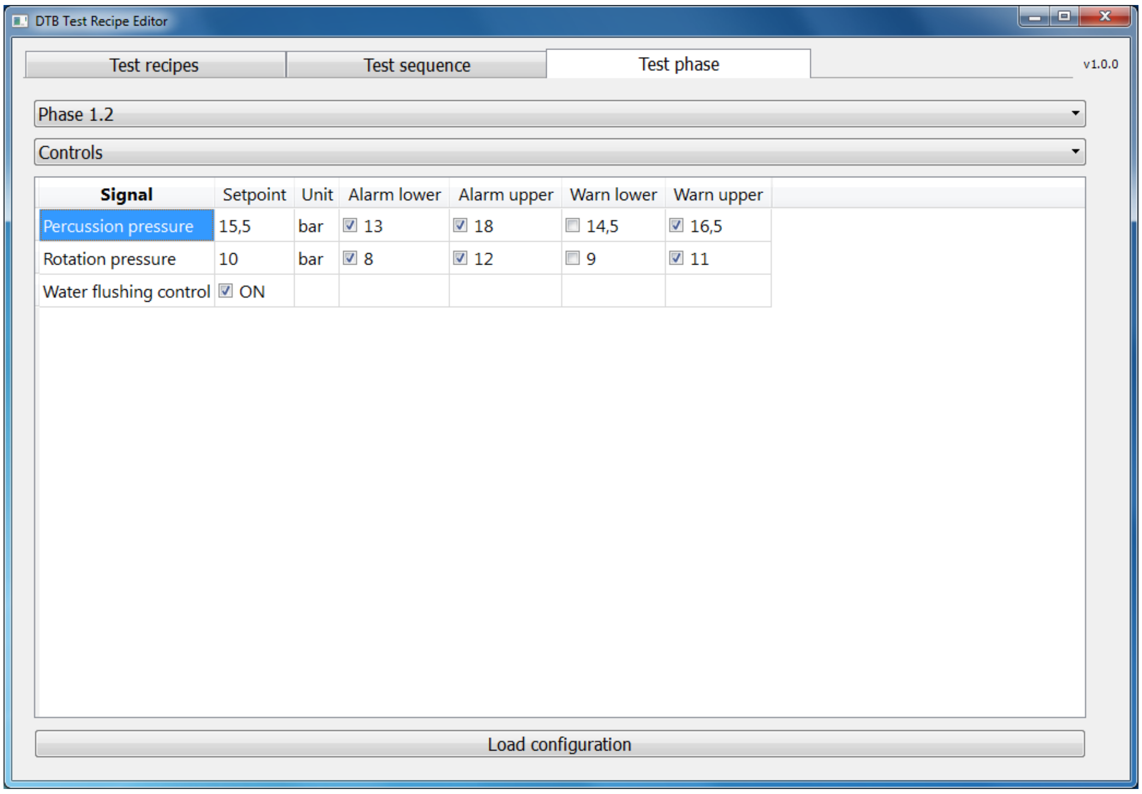
Signal	Unit	Alarm lower	Alarm upper	Warn lowe
Tank oil temperature	C	☑ 60	☑ 80	☐ 0

Tank oil temperature

Test recipe: Example 1

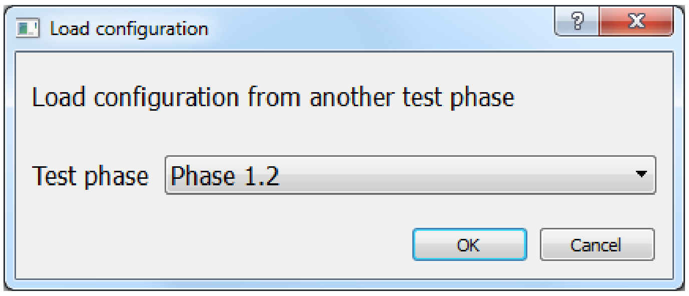
		Lower limit	Upper limit	
Alarm limits	☑	60	80	☑
Warning limits	☐	0	95	☑

Figure 4.8. The test phase view when editing a floating point type measurement signal.



Signal	Setpoint	Unit	Alarm lower	Alarm upper	Warn lower	Warn upper
Percussion pressure	15,5	bar	✓ 13	✓ 18	☐ 14,5	✓ 16,5
Rotation pressure	10	bar	✓ 8	✓ 12	☐ 9	✓ 11
Water flushing control	✓ ON					

Figure 4.9. The test phase view with the table expanded to full width.



Load configuration from another test phase

Test phase: Phase 1.2

Figure 4.10. The dialog that is shown when the "Load configuration" button is pressed from the test phase view.

5. DURABILITY TESTING OF ROCK DRILLS WITH A TEST BENCH

Sandvik Mining and Construction Oy manufactures drill rigs, which are machines used for rock drilling. One machine can have one or several rock drills, which can be used to drill several holes simultaneously. Both the drill rigs and the rock drills can be tested in test benches. This thesis deals with test benches used for testing rock drills, and testing of drill rigs is left outside the scope of this thesis. For more information on drill rig testing with a test bench, see theses [12] and [9].

Section 5.1 gives an overview of different kinds of test benches that are used for testing rock drills. Testing methods and needs of this DTB project are described in Section 5.2. The architecture of the control system is discussed in Section 5.3. Finally, Section 5.4 presents the human-machine interface of the DTB.

5.1 Overview of drill test benches

Sandvik uses four kinds of test benches for rock drill testing. The overview given in this section is based on an interview with Tomi Nieminen from Sandvik [1].

The *research team* uses DTBs to test new ideas and technologies with prototype drills. The main focus is on testing performance by measuring the penetration velocity of a drill. The research unit uses either a rock or a *cushion rock drill* in their DTBs. A cushion rock drill absorbs the percussion pulse normally going to the rock from the percussing rock drill and transfers the energy to hydraulic oil [2]. It can be used to simulate rock drilling, but real conditions can only be achieved by drilling into rock. The drill is controlled manually by using switches and potentiometers. Since needs of the research change all the time, full manual control needs to be available in these test benches, and their control systems need to be adaptable to changes.

The *product development team* develops new products by using old designs and well-known technologies as a basis and making only minor enhancements to them. The purpose of the DTBs used by the product development team is to test the durability of a new drill in the long run and expose possible flaws in the design by investigating what damages result from a long-term operation. If the new drill does not show weaknesses or excessive wearing, production of the drill can be started. The drills are tested by using automatic tests that drive the drill towards a cushion rock drill. Manual control is needed for attaching the drill to the drill equipment and for detaching from it. Needs of the testing are relatively well known since it is known which kinds of drills are tested in the DTB. Therefore, development of a control system with long life cycle and less adaptability than in the research DTBs is possible. Although each team has their own DTBs, the product development team occasionally uses also DTBs of the research and production units.

The *production team* uses DTBs for assuring quality of every drill unit that is produced. The tests are automatic and short enough so that the drill does not wear. The DTB performs measurements and monitors that they are within specified limits. If a limit is exceeded, the unit may be faulty and needs investigation. Otherwise, it is accepted to be delivered to a customer. Manual control is used for attaching and detaching the drill. For more information on DTBs used by the production, see thesis [13].

Sandvik has servicing points in the field where drills are overhauled. The *service team* uses DTBs to verify correct operation and expected performance of an overhauled drill. The drill is returned to the overhaul if it does not pass the tests. The DTBs are similar to the ones used in the production. For more information on DTBs used by the servicing points, see theses [25] and [15]. The prototype DTB built in the project described by [25] has later been used by the product development team to perform their testing.

5.2 Testing methods and needs

In this project a control system is developed for DTBs that would be used by the product development team for durability testing. The aim of this testing is to get a large number of operating hours for a drill under test in considerably less time than what would be possible if the testing was performed in the field with a drill rig. A drill is tested for 24 hours at a time; however, the test may be interrupted sooner by a breakage in the drill equipment. Overall, testing of a new drill product may take 2–3 months. [1]

One essential feature for durability testing, where test runs are long, is the possibility to have cyclic control of certain signals. Previously this has been achieved by defining a constant control signal in the test recipe and performing the cycling with hardware such as an electric timer. In this project the test recipes include a possibility to define test loops so that the cycling can be done with software. [1]

Since the tests are automatic, the operation of the drill needs to be monitored. Two kinds of monitoring limits can be defined for measurement signals. Exceeding alarm limits implies that the test has failed and needs to be stopped immediately. This may be caused by, for example, a loose hose causing a leakage of hydraulic oil and a pressure drop. Exceeding warning limits implies that the quantity is not within normal, expected range, but the test need not be stopped. Both alarms and warnings are recorded in an alarm log. The log is saved along with other test results. The alarm log can be checked later to determine the reason why a test was stopped. However, the real problem, the cause for exceeding a limit, may be different. Therefore, the measurement signals need to be logged. It is important to get detailed data with high sampling rate from the last ten seconds before the test was stopped. Since large amounts of data is difficult to handle, rest of the test data is stored with lower sampling rate. [1]

5.3 Control system architecture

The control system of the DTB is based on SICA (Sandvik Intelligent Control Architecture). SICA based control systems are divided into three levels as shown in Figure 5.1: *the supervisor level (SUP)*, *machine control level (MC)*, and *safety level*. The SUP level handles functions that are not safety critical and have no hard real-time requirements, such as graphical user interfaces and data acquisition. It consists of one or more supervisor modules which are usually industrial PCs with a display, either a touch screen or a display with buttons. The MC level is responsible for the actual control functions in safe, predictable, and fault tolerant way. Application projects may choose to use either *Calculation Units (CU)* with the SICA MCC platform, or PLCs with the IEC (International Electrotechnical Commission) code template, or both. Depending on the selected hardware, there may be I/O (Input/Output) in the CU or PLC in itself, but additional I/O modules that communicate via CAN bus can also be used. The safety level implements SIL (Safety Integrity Level) compliant functions such as emergency stop. The safety level is implemented with SILx certified safety PLCs and hardwired electrics and hydraulics, and it contains no application software. [38, p. 23]; [35, pp. 7, 10]

In this project only the SICA MCC platform is used, and the IEC platform is left outside the scope of this thesis. The SUP level is described in Subsection 5.3.1. CUs and I/O modules of the MC level are described in Subsections 5.3.2 and 5.3.3, respectively. The safety level is not software related and is therefore not described in more detail in this thesis.

5.3.1 Supervisor

A supervisor is the highest level component of the control system. In this project an industrial PC is used as a SUP module, and it has a 12-inch touch screen and a resolution of 1024x768 pixels. The PC has CAN and Ethernet connectivity, and the communication to CU is done via Ethernet. The PC runs a Linux operating system. Real-time threads are not used since the supervisor level is not meant to be used for hard real-time functions.

SUP runs an application written in Qt/C++ on top of the SICA SUP platform. Tasks and features of the application include displaying diagnostics, adjusting parameters, recording the alarm log, data acquisition, editing test recipes, and running the test engine. Editing test recipes is performed with the editor discussed in Chapter 4. The test engine implementation will be discussed in more detail in Chapter 6, which will also show a

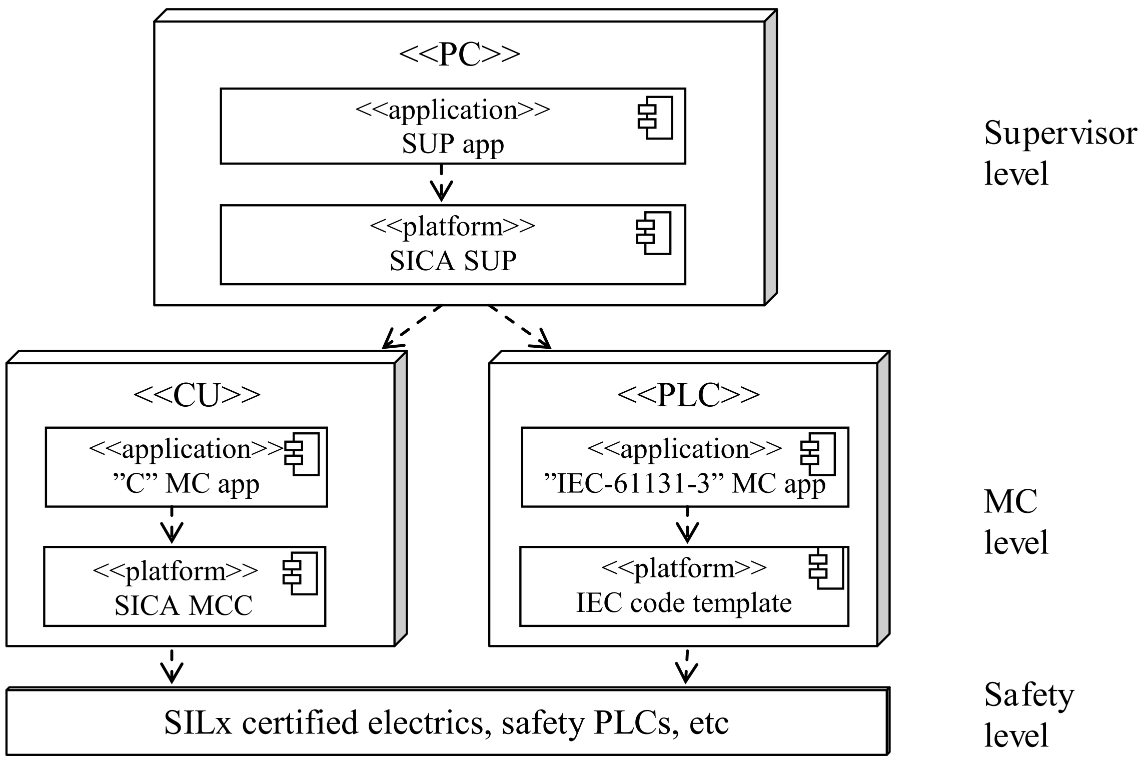

<<PC>>
<<application>> SUP app
<<platform>> SICA SUP
Supervisor level
<<CU>>
<<application>> "C" MC app
<<platform>> SICA MCC
<<PLC>>
<<application>> "IEC-61131-3" MC app
<<platform>> IEC code template
MC level
SILx certified electrics, safety PLCs, etc
Safety level

Figure 5.1. *Levels of SICA based control systems. The arrows indicate dependencies between components. The figure has been modified from the figure in source [35, p. 7].*

screenshot of the test recipe editor running in the SUP display. Descriptions of the other features of the application are left outside the scope of this thesis.

5.3.2 Calculation unit

A calculation unit is a component where the hard real-time functions of the control system are run. In the DTB a Linux-based module with CAN and Ethernet connectivity is used. Figure 5.2 shows how modules are connected to each other when using the SICA MCC platform. CU communicates with SUP via Ethernet and with I/O modules via CAN bus. The SICA MCC platform is very flexible in terms of the system architecture. There can be several CUs, and on the other hand, several MC applications can be run in one CU. In the DTB one CU is used since there is currently no need for more.

The CU runs two MC applications written in C language on top of the SICA MCC platform. An application called FRQC controls a frequency converter. Another application called DTB controls all other actuators, monitors sensors, and raises alarms if limits are exceeded. In manual control mode it follows the commands given by an operator using a *radio remote controller (RRC)*. In automatic control mode it follows the commands given by the test engine of the SUP module.

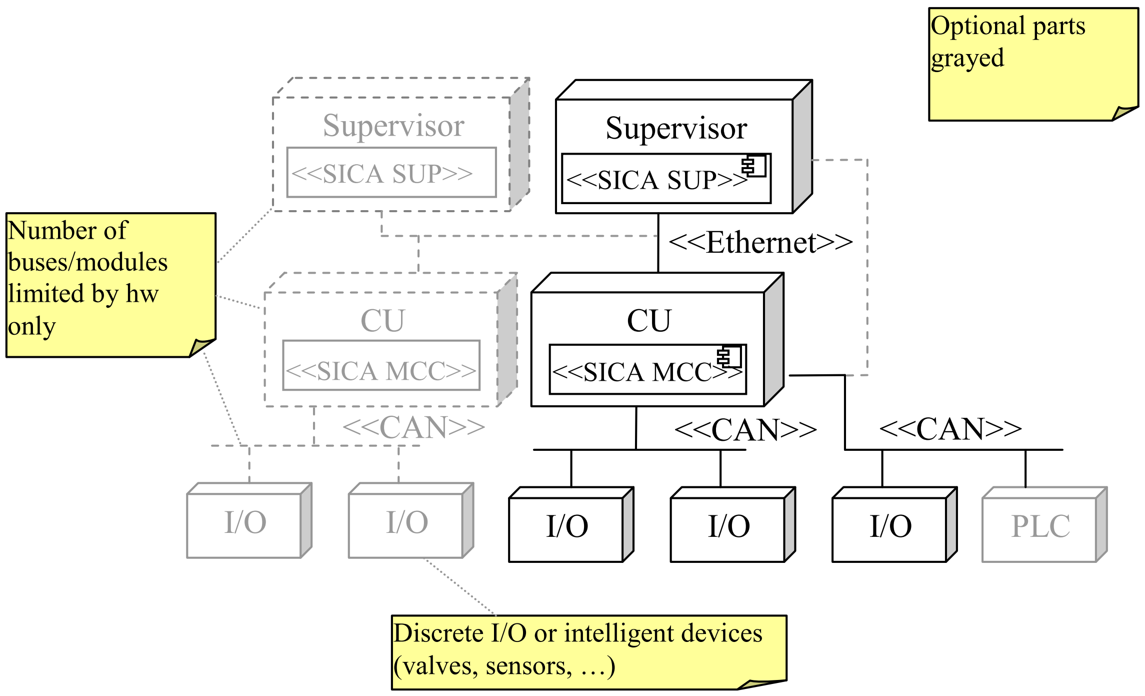


Figure 5.2. *The system architecture when using the SICA MCC platform. The figure has been modified from the figure in source [35, p. 10].*

5.3.3 I/O modules

I/O modules are modules where actuators and sensors are connected. The DTB uses several I/O modules, which communicate via CAN bus using CANopen protocol. The modules are configured automatically by SICA based on configurations defined in the DTB project.

Additionally, there are two other devices in the system: a frequency converter and a receiver for an RRC. They are CANopen devices and are configured automatically by SICA.

5.4 Human-machine interface

The human-machine interface of the DTB, located in a cabin, is illustrated in Figure 5.3. Some aspects like safety functions of the DTB are not described in this thesis; therefore, some parts of the HMI outside the cabin are irrelevant in this thesis and are not shown. In addition to the SUP display, the cabin contains a control panel and a rack for an RRC. The control panel has buttons and switches that are used to control power and state of the DTB. The RRC can be used to manually control the DTB. Manual control is used for attaching and detaching the drill under test.

The operator needs to do the following sequence to start manual control: First, the main power switch is turned on, which powers the control system on. The auto/manual

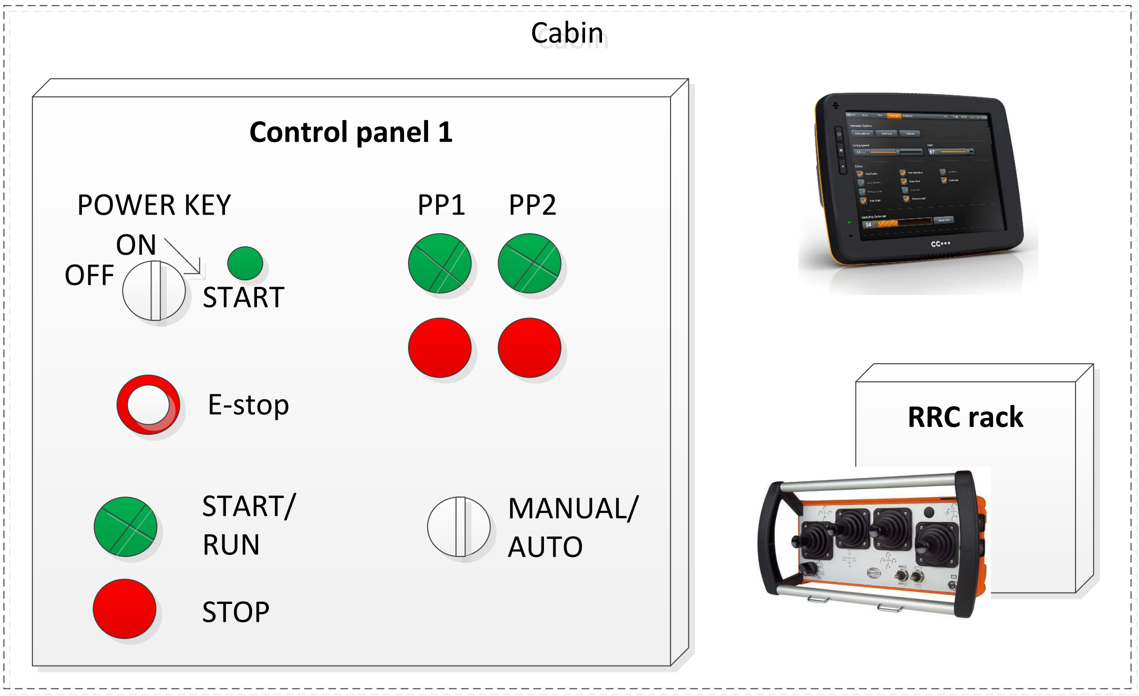


Figure 5.3. *The human-machine interface of the DTB. The images shown are indicative only. The figure has been modified from the figure in source [36].*

mode switch needs to be in the “MANUAL” position. The RRC is then turned on and can be taken out from its rack if desired. The power key is turned to the “START” position, which starts the machine. The power packs are started with the “PP1 START” and “PP2 START” buttons. The DTB is now ready to be controlled with the RRC.

Automatic tests are performed in the following way: First, the main power switch is turned on, which powers the control system on. The auto/manual switch needs to be in the “MANUAL” position. After the SUP display has booted, a USB (Universal Serial Bus) stick is connected to the display, and test recipes are loaded from it. A test recipe is selected to be run by pressing the “Run” button in the SUP GUI. The power key is turned to the “START” position, which starts the machine. The RRC needs to be in its rack, but power can be off. The power packs are started with the “PP1 START” and “PP2 START” buttons. The mode switch is switched to the “AUTO” position. Then, the “RUN” button is pressed, which starts the automatic test sequence. After the test has finished, the CU lets the power packs run for a while prior to shutting them down. A new test run can be started by closing the finished test run from the SUP GUI, selecting a test recipe to be run and pressing the “Run” button, starting the power packs from the control panel, and pressing the “RUN” button in the control panel.

6. DEVELOPMENT OF A TEST ENGINE

This chapter describes the implementation of a test engine made in the customer project. In addition to the test engine itself, a GUI is needed for starting and monitoring a test run. Design of the implemented test engine is discussed in Section 6.1. Technology choices and other details of the implementation are described in Section 6.2. The GUI of the test engine is described and depicted in Section 6.3.

6.1 Design

The test engine implementation is based on what was outlined in the testing framework concept in Section 3.4. However, it differs from the concept in that the SUP level implementation does not monitor the output signals. It is the responsibility of the MC level to monitor the signals and raise alarms if verification criteria are not met. Status of the MC level is reported with one signal which indicates whether running a test is possible or not. The test engine listens to this signal, and if it goes low during a test run, indicating a test failure, the test engine will stop running the test and report the failure to the user.

Response latencies and accuracy in measuring time are not critical in the test engine implementation. If the user presses an emergency stop button, the DTB must stop immediately, but it is the MC level which handles this. The test engine need not react as rapidly as the MC since the only effect is a minor latency in updating the user interface. Test runs can be tens of hours long, and the resolution of a test phase duration is one second, so achieving a highly accurate test duration is not critical. Updating all signals exactly at the same moment when a test phase is changed is not critical either, because the response of the mechanics is much slower than the latencies in the control system. Due to these reasons it is sufficient to run the test engine with a one second period.

The user is informed of the status of the test run in the user interface. This includes displaying the total duration of the running test, time left until the test is finished, and the currently running test step. Additionally, the user has the opportunity to write notes regarding the test run either during the test run or after it has finished.

6.2 Implementation

As in the case of the test recipe editor, the test engine was implemented with Qt 4.7.1 and C++. Additionally, SICA SUP platform services were used. The package diagram

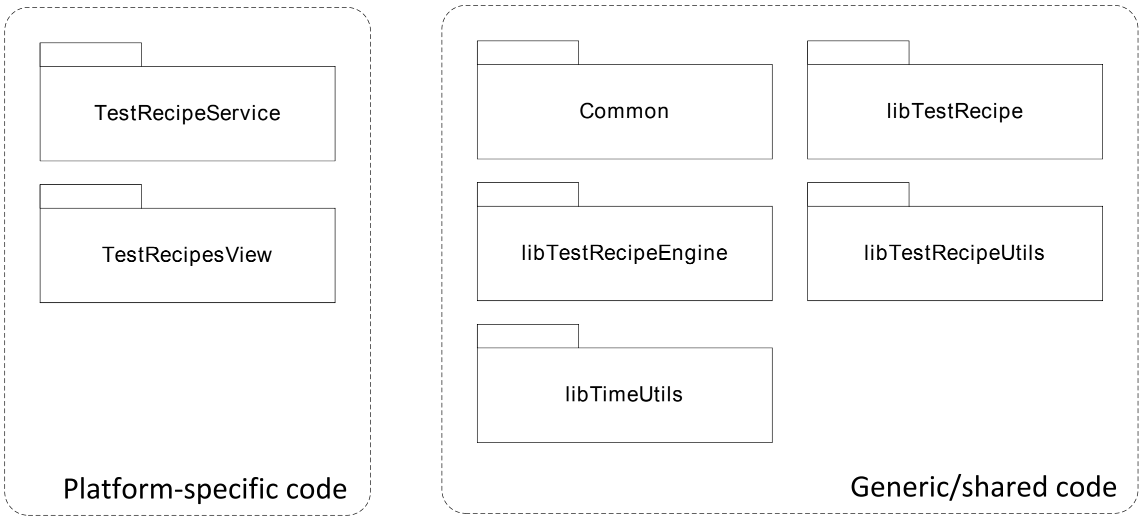


Figure 6.1. *The package diagram of the test recipe engine.*

of the test engine is depicted in Figure 6.1. The implementation consists of a generic part and a SICA dependent part. The generic part is implemented in a library called `libTestRecipeEngine`. This library implements the logic related to running one test run cycle including changing the test step when an end condition is met and writing signals. The library defines interfaces for signal communication and reporting progress of the test run, which need to be implemented by the client of the library. The library provides a test run creator which returns an interface used for running the test one cycle at a time with given elapsed time after previous run cycle. The test recipe data structure is used from the shared `libTestRecipe` library.

The other part of the test engine is called `TestRecipeService`. It implements the *SICA service interface*. SICA Starter loads and starts the service and registers it to SICA ServiceLocator. The SUP application can fetch the service specific `ITestRecipeService` interface from ServiceLocator. [38, p. 116] This interface provides services for the test engine GUI. `TestRecipeService` implements the interfaces defined by `libTestRecipeEngine` and executes a test run by using a timer. It utilizes the packages `Common`, `libTestRecipeUtils`, and `libTimeUtils`, described earlier in the test recipe editor implementation, in its implementation. The test engine GUI is implemented in the `TestRecipesView` library.

Communication between the SUP and MC levels is done by means of signals. In the SICA platform signal communication is handled by a software library called MCon. According to the MCon user manual, MCon aims to abstract away any field bus, protocol, bus layout, and hardware specific details. It abstracts the control system to a set of signals that can be accessed through a generic API. [37, pp. 10-11] In addition to the signals defined in the signal interface for the test recipes, there are a few signals that the test engine reads and writes to control the MC level. These signals are described in Table 6.1.

Table 6.1. *Signals which the test engine uses to communicate with the MC level.*

Signal name	Access	Purpose
AutoMode-Monitoring-Delay_ms	Write	Specifies a delay for the monitoring to be enabled when moving to a next test step. The delay is used to prevent false alarms when control setpoints are changed and the control response has not yet reached the level where it would be within the new expected limits.
AutoModeStep	Write	A counter for the test steps. A change in this signal indicates change to the next test step. Prior to starting a test, the signal value is zero. After a test run has finished, the signal value is -1.
AutoRun	Read	A Boolean type signal which indicates if running a test is possible. The test engine starts execution after MC raises this signal. If MC drops the signal during a test run, it indicates that the test run has failed or it has been aborted by the user.
SUPHB	Write	The heartbeat of the test engine. If the heartbeat signal value has not changed during the last five seconds, MC stops the test and goes to error state. This guarantees that the test is stopped if, for example, the SUP software crashes or there is a problem in communication between SUP and MC.

The test engine is run periodically with a one second timer, which is implemented with the Qt's `QTimer` class. Qt cannot guarantee that the timer will trigger exactly as specified since the accuracy of timers depends on the underlying operating system and hardware [29]. Therefore, the actual elapsed time between two run cycles is calculated with the Qt's `QElapsedTimer` class. Possible error in timer cycle is thus taken into account and will not cumulate to make any significant error in the long run.

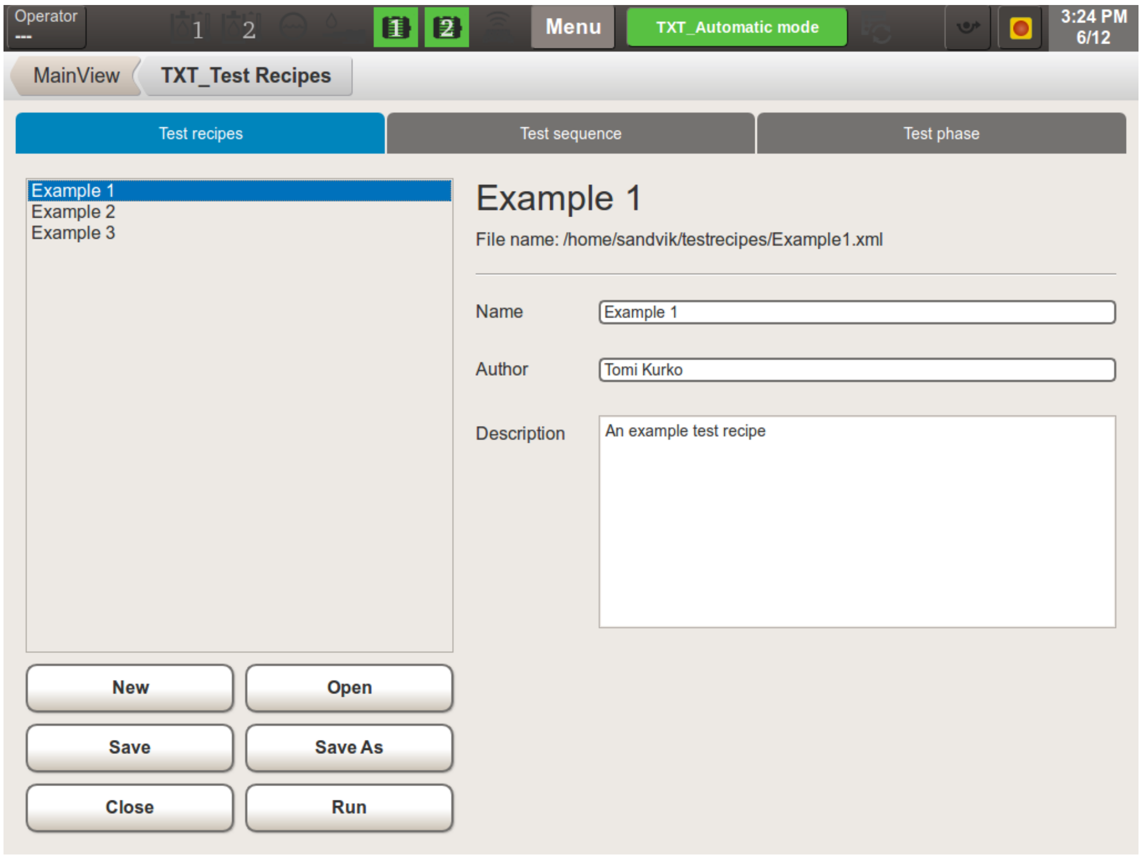
6.3 User interface

The SUP level user interface has two views that are related to the test engine: *the test recipe selection view* and *test recipe run view*. They are presented in Subsections 6.3.1 and 6.3.2, respectively. Only one of the views is available at a time depending on the state of the test engine.

6.3.1 Test recipe selection view

The test recipe selection view is where the user selects which test recipe they want to run. Selection is possible only when the test engine is not running a test and no test recipe has been selected for running. This is done to prevent the user from modifying a test recipe while it is being executed. Modification could be allowed by making the implementation such that the changes would not affect the running test, but it is better to avoid any possible confusion.

Selection of a test recipe to be run is done in the test recipe view of the test recipe editor, which is shown in Figure 6.2. The view is equal to the desktop version except that it has the SICA appearance and it has a “Run” button. Pressing the “Run” button selects the currently active test recipe to be run by the test engine. This does not, however, start the execution yet. It only marks the recipe as selected, and the view is changed to the test recipe run view.



Example 1

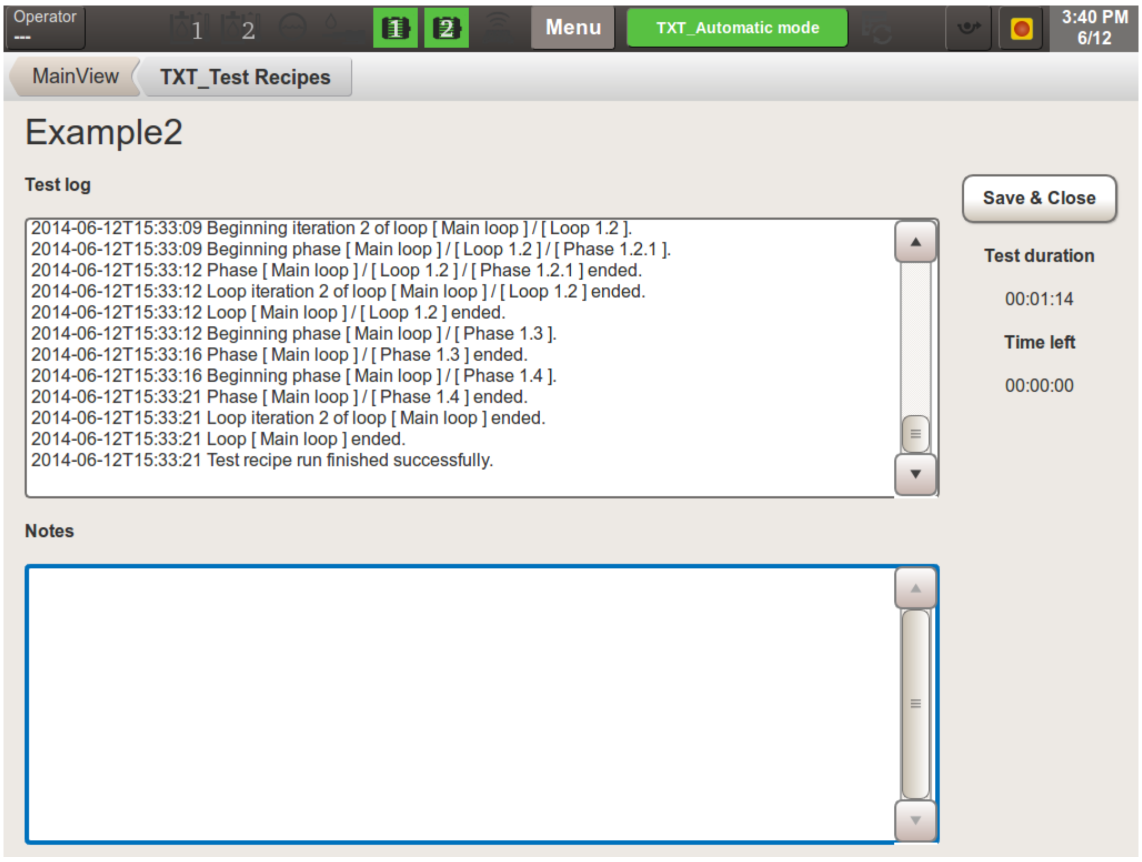
File name: /home/sandvik/testrecipes/Example1.xml

Name	Example 1
Author	Tomi Kurko
Description	An example test recipe

Figure 6.2. The test recipe view of the test recipe editor running in the SUP display. In addition to the editor functionalities that exist in the desktop version, the view has a “Run” button, which is used to select a test recipe for running.

6.3.2 Test recipe run view

The test recipe run view shows the status of the test recipe run. The view is shown in Figure 6.3. It shows the progress of the test by showing how much time is left and what is the total duration of the test. A test log with timestamps is shown where beginning and ending of test steps is reported. Success or a reason for a failure is also reported when the test is finished. During and after the test run the user may write notes in a text box.



Example2

Test log

```
2014-06-12T15:33:09 Beginning iteration 2 of loop [ Main loop ] / [ Loop 1.2 ].
2014-06-12T15:33:09 Beginning phase [ Main loop ] / [ Loop 1.2 ] / [ Phase 1.2.1 ].
2014-06-12T15:33:12 Phase [ Main loop ] / [ Loop 1.2 ] / [ Phase 1.2.1 ] ended.
2014-06-12T15:33:12 Loop iteration 2 of loop [ Main loop ] / [ Loop 1.2 ] ended.
2014-06-12T15:33:12 Loop [ Main loop ] / [ Loop 1.2 ] ended.
2014-06-12T15:33:12 Beginning phase [ Main loop ] / [ Phase 1.3 ].
2014-06-12T15:33:16 Phase [ Main loop ] / [ Phase 1.3 ] ended.
2014-06-12T15:33:16 Beginning phase [ Main loop ] / [ Phase 1.4 ].
2014-06-12T15:33:21 Phase [ Main loop ] / [ Phase 1.4 ] ended.
2014-06-12T15:33:21 Loop iteration 2 of loop [ Main loop ] ended.
2014-06-12T15:33:21 Loop [ Main loop ] ended.
2014-06-12T15:33:21 Test recipe run finished successfully.
```

Test duration

00:01:14

Time left

00:00:00

Figure 6.3. *The test recipe run view.*

Figure 6.3 presents the view when the test run has finished. At this stage there is a button called “Save & Close” which saves the notes to the test results and changes the view to the selection view. Before a selected test is started, this button is named as “Cancel”. Pressing this button cancels the selection of a test recipe and takes the user back to the selection view. Execution of a test starts only after a recipe has been selected for running and the user presses the mechanical run button in the DTB’s human-machine interface. During a test run the “Cancel” button is disabled and stopping is possible only by pressing the mechanical stop button or an emergency stop button.

7. EVALUATION AND FURTHER DEVELOPMENT

This chapter discusses evaluation of the new control system and what further development could be done. The evaluation is presented in Section 7.1. Possible extensions for test recipe definitions are discussed in Section 7.2. Section 7.3 describes the current state of the implementation of touch input support.

7.1 Evaluation of the new control system

Due to the cancellation of the planned DTB projects, the outlook for deploying the new control system is currently unknown. Consequently, no feedback on real use experiences was received during the process of writing this thesis. However, the new control system is evaluated by comparing it to one used in an older DTB (see [25]).

The basic idea of having test recipes derives from older DTBs, so the concept of testing rock drills with DTBs remains the same. There are, however, some improvements over the old DTB, which make the user experience better. Looping of test phases is easier since it can be done in test recipes instead of using electric timers. It also makes test recipes more self-documenting. Other improvements to documenting test recipes include having names and descriptions for test steps instead of plain numbers. In the old DTB software, there is a fixed set of signals used in all test phases, but each test phase may also have a set of additional signals that are not included in the fixed signal list. Creating test recipes for the old software also requires undocumented knowledge of how the signals in these different lists are handled by the software so that the result would be correct. Thus, using the software is not intuitive, which makes the usability poor. The new software uses a fixed set of signals in all test phases, which is simpler and easier to understand. Although the test recipe editor has not been in production use yet, according to the customer it seems relatively simple and convenient to use when compared to software in some other DTBs. [1]

Although the improvements in user experience are significant, the most considerable reason for developing this new control system was to achieve better maintainability. The old software and hardware were not easily maintainable anymore. The use of the SICA platform and hardware components from the Sandvik's hardware library results in better maintainability since there is better internal knowledge of these components inside Sandvik. There was also a desire to use this control system in several forthcoming DTBs so that the overall maintenance effort of all DTBs would be reduced. [1]

The old DTB software is a Windows GUI application run on an ordinary PC with a CAN interface adapter. This software has included not only a test recipe editor but also most of the control logic that is needed to control the DTB. This involves, for example, control feedbacks. Since PC hardware cannot provide real-time functions, the feedbacks have been slow, which has caused problems. The new control system runs the feedbacks in a hard real-time system, so this should not be an issue anymore. [1]

The old DTB software uses a database to store test recipes and results of test runs, whereas the new software uses files to store this data. Both approaches certainly have advantages and disadvantages, but the file approach may be somewhat easier to maintain. The database has been situated locally in the PC running the software, so there has been no need, for example, for synchronizing test recipes and test results to a central database. [1]

7.2 Extensions for test recipe definitions

The testing framework that was implemented in the customer project did not include all of the features that were explained in the testing framework concept. This section lists some improvements that could be implemented to make the testing framework more versatile and suitable for various forms of testing.

Modularized test recipes: Test recipes could include other test recipes or fragments of them by having references to them. This would make it possible to maintain smaller fragments which would be imported to several test recipes. Then, test recipes could be defined even by having only references to smaller fragments. However, it depends on the application domain whether there is use for this kind of feature. In the DTBs there is currently no recognized need for this feature. The implemented test recipe editor currently supports only loading signal configurations to a test phase from another test phase in the same test recipe.

More input signal function types: In the implemented testing framework, only constant input signal functions are supported. Many more function types could be supported. For instance, linear ramps could be useful in many applications of testing. In the beginning of the DTB project, linear ramps were discussed as a possible feature that might be required, but it was given a low priority and was not yet implemented.

More output signal verification criteria: In the implemented testing framework, only the “Range” criterion is supported for verifying output signals. It is used for monitoring analog measurement signals. Many more verification criteria could be supported. For instance, if linear ramps were implemented, they could be verified with the “Gradient” criterion. The “Value” criterion could be used to verify digital signals. Monitoring of digital signals was discussed in the DTB project, but it was not seen necessary at this point.

Definition of monitoring limits by tolerance: In the implemented testing framework, monitoring limits are defined by specifying a lower and upper limit. It was discussed that in most cases it would be sufficient and easier to define the limits by specifying a tolerance. In the case of control signals, this means that the tolerance defines how far from the setpoint the measured value is allowed to be without raising an alarm or warning. In the case of measurement signals, an expected value would need to be defined in addition to a tolerance. The tolerance method would be useful especially for defining the monitoring limits of control signals. This feature was seen as relatively important since it would improve usability, but it was not yet implemented.

More test step end conditions: It was discussed in the testing framework concept in Section 3.2 that test phases and loops could have end conditions that are Boolean expressions dependent of signal values at run-time and elapsed time or loop iterations. The Boolean expressions could include operations of Boolean algebra such as conjuction (AND), disjunction (OR), and negation (NOT). This would enable the test designer to define tests whose progress depends on real-life conditions. For instance, if drilling was performed into rock instead of using a cushion rock drill, an end condition could be that a certain drilling depth has been achieved. This condition could also be OR'ed with a condition that certain time has elapsed. This would prevent the test from getting stuck in a test phase if the sensor measuring drilling depth was broken. Using drilling depth as an end condition was discussed briefly in the DTB project, but it was not included in the requirements since the first DTBs that would use the new control system would not use it.

Dependent signals: Instead of defining all control signals by numerical values, they could be defined as functions of other control signals. This would make test recipes more descriptive in terms of their intention and also make their maintenance easier. In the original specification of the DTB software [24], there was a mention regarding *feed-percussion follow up*, which means that percussion pressure is adjusted according to the feed pressure used [2]. This implies that the percussion pressure control signal would be defined as a function of the feed pressure measurement signal. This, in turn, implies that the control signal is dependent on run-time behaviour of the DTB. The test recipe may thus control the DTB differently on different test runs. This functionality might be needed in a DTB designed for the research team, who perform real drilling into rock. In DTBs using a cushion rock drill, feed-percussion follow up is not needed. The relation between the dependent signals could be defined by specifying a slope which defines how much a signal changes as a function of another signal. In addition to the slope, minimum and maximum values could be defined for the signal that is a function of another signal. [1]

7.3 Touch input support

The SUP display that was used in the DTB project is a touch screen. All GUIs were designed to be usable with touch input. This involved making the controls large enough so that they can be easily touched but also selecting SICA widgets that support touch input. In practice, widgets that accept text or numerical input need to show a virtual keyboard on the screen.

Originally, touch input support was in the requirements, which affected the GUI design. Later, it was discussed that writing long texts with an on-screen keyboard is awkward, and users prefer using a keyboard and mouse. Since the SUP module supports USB devices, it was decided that a keyboard and mouse would be attached to it. Thus, implementing touch input support was set to a lower priority. The two branches, desktop and SUP versions, were still kept, but touch input support was not finished. Consequently, some controls of the GUI are not usable with touch input. However, all controls are usable with a keyboard and mouse.

8. CONCLUSIONS

In this thesis a new control system was developed for test benches that are used for testing durability of new rock drill products prior to starting their production. The DTBs are used by the product development team at Sandvik Mining and Construction Oy, which manufactures the rock drills and the drill rigs where the rock drills are used. In addition to product development, Sandvik uses DTBs in research, production, and in their servicing points in the field. DTBs are mainly used for assuring quality of manufactured products.

A concept of a testing framework was designed which describes conceptually how the automatic tests are defined and run. The concept is based on a testing methodology called signal based testing. Signal based testing relies on a generic signal interface between the tests and the system under test. The tests are called test recipes, which consist of test phases and loops. Applicability of the concept was analyzed also in a broader context, and it was concluded that the concept is mostly suitable for testing scenarios similar to rock drill testing; that is, testing performance and reliability of mechanical, hydraulic, and electrical devices. Additionally, it could be useful in testing some embedded and distributed control systems.

Development of the new control system was started from scratch but on top of the Sandvik's software platform and system architecture called SICA. The software of a previous DTB was used as a basis for designing the test recipe editor. New features were implemented in the editor that improve usability and bring better support for durability testing. The implementation of a test engine, which executes test recipes, and related GUI were also described. Other software components of the control system were discussed only briefly since they were not implemented by the author. The most significant advantage of the new control system is better maintainability due to good internal knowledge at Sandvik of the hardware components used and better control over the development of the control system. The aim was also to unify DTB control systems by using the new control system in several DTBs to reduce overall maintenance costs.

The project plans for building new DTBs were cancelled during the process of writing this thesis, but it was decided that development of the new control system would be finished. Therefore, feedback from production use could not be received. However, according to current assessment, the project has achieved its goals, and no major issues were encountered. The test recipe editor was tested thoroughly with manual tests. The test engine and related GUI were tested with the MCC application and a simulator. Additionally, some tests were written for the test engine.

Currently, there is no outlook for when the new control system will be deployed in a DTB. It can be used in new DTB projects or when an old DTB control system is updated. When the deployment is done, further testing needs to be done with the real hardware. Depending on the requirements of the DTB where the control system is to be deployed, further development may be needed to support additional features in the test recipes.

REFERENCES

[1] Interviews with Tomi Nieminen (a Sandvik employee) on 9th May 2014 and 9th June 2014.

[2] SMC Dictionary. Technical report, Sandvik Mining and Construction. Updated on 2nd January 2014. Company Confidential.

[3] IEEE Standard for Signal and Test Definition. *IEEE Std 1641-2010 (Revision of IEEE Std 1641-2004)*, pages 1–334, Sept 2010.

[4] Panu Ahola. Production testing of distributed control systems in mobile mining machines. Master's thesis, Tampere University of Technology, 2009.

[5] Karl Johan Aström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2010.

[6] R.D. Craig and S.P. Jaskiel. *Systematic Software Testing*. Artech House computing library. Artech House, 2002.

[7] M. Fewster and D. Graham. *Software test automation: effective use of test execution tools*. ACM Press books. Addison-Wesley, 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[9] Heikki Heiskanen. Testausympäristön suunnittelu ja käyttöönotto. Master's thesis, Tampere University of Technology, 2006.

[10] Cory Janssen. Automatic Test Equipment (ATE). `http://www.techopedia.com/definition/2148/automatic-test-equipment-ate`. Retrieved on 9th March 2014.

[11] Dag Kristiansen and Karl-Petter Lindegaard. A framework for automatic testing of industrial controller code. In *Software Engineering Research and Practice*, pages 3–9. Citeseer, 2006.

[12] Simo Lahtinen. Kallioporakoneen testipenkin ohjaus. Master's thesis, Tampere University of Technology, 2008.

[13] Martti Leskinen. Hydraulisten iskulaitteiden testijärjestelmän suunnittelu. Master's thesis, Tampere University of Technology, 2009.

[14] K.-P. Lindegaard and D. Kristiansen. Signal based functionality testing of control systems. In *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE*, pages 2635–2640, Oct 2006.

[15] Jukka Luutikivi. Iskulaitteen testipenkin käytettävyyden kehittäminen. Bachelor's thesis, Tampere University of Applied Sciences, 2011.

[16] W. Mahnke, S.H. Leitner, and M. Damm. *OPC Unified Architecture*. SpringerLink: Springer e-Books. Springer, 2009.

[17] Jason McDonald. Qt 4.7.1 Released. `https://blog.qt.digia.com/blog/2010/11/09/qt-4-7-1-released/`. Retrieved on 10th June 2014.

[18] G.J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. ITPro collection. Wiley, 2011.

[19] National Instruments Corporation. Application Areas: Where Is NI TestStand Used? `http://www.ni.com/teststand/applications/`. Retrieved on 9th March 2014.

[20] National Instruments Corporation. Automated Test Application Areas. `http://www.ni.com/automatedtest/applications/`. Retrieved on 9th March 2014.

[21] National Instruments Corporation. Semiconductor Test. `http://www.ni.com/automatedtest/semiconductor/`. Retrieved on 9th March 2014.

[22] National Technical Systems, Inc. Product Reliability Testing. `http://www.nts.com/services/dynamics/reliability`. Retrieved on 5th June 2014.

[23] NexLogic. In Circuit Testing (ICT). `http://www.nexlogic.com/services/pcb-testing/ict-testing.aspx`. Retrieved on 5th June 2014.

[24] Tomi Nieminen. Mjöllner ja XL Ohjausjärjestelmän ja käyttöliittymän määrittely v1.0. Technical report, Sandvik Mining and Construction. Updated on 8th October 2013. Company Confidential.

[25] Tomi Nieminen. Hydraulisten iskulaitteiden koekäyttöjärjestelmä. Master's thesis, Tampere University of Technology, 2008.

[26] Jani Pesonen, Mika Katara, and Tommi Mikkonen. Production-testing of embedded systems with aspects. In *Hardware and Software, Verification and Testing*, pages 90–102. Springer, 2006.

[27] Qt Project Hosting. Model/View Programming. `http://qt-project.org/doc/qt-4.7/model-view-programming.html`. Retrieved on 28th March 2014.

[28] Qt Project Hosting. Qt Project. `http://qt-project.org/`. Retrieved on 28th March 2014.

[29] Qt Project Hosting. QTimer Class Reference. `http://qt-project.org/doc/qt-4.7/qtimer.html`. Retrieved on 25th April 2014.

[30] Qt Project Hosting. User Interface Compiler (uic). `http://qt-project.org/doc/qt-4.7/uic.html`. Retrieved on 4th May 2014.

[31] Sandvik AB. About Sandvik. `http://www.sandvik.com/en/about-sandvik/`. Retrieved on 19th April 2014.

[32] Sandvik Mining and Construction Finland Oy. Sandvik Mining and Construction Finland Oy. `http://www.miningandconstruction.sandvik.com/fi`. Retrieved on 19th April 2014.

[33] Andy Shaw. Qt 4.7.2 has been released! `http://blog.qt.digia.com/blog/2011/03/01/qt-4-7-2-has-been-released/`. Retrieved on 10th June 2014.

[34] Marc van't Veer. What is testing in production and why should it be performed? *Testing Experience*, (20):48–51, Dec 2012.

[35] Janne Viitala. SICA – MC level architecture. Technical report, Sandvik Mining and Construction. Revision 0.9, Updated on 26th May 2009. Company Confidential.

[36] Janne Viitala. XL-penkki käyttö ja turvallisuuskonsepti. Technical report, Sandvik Mining and Construction. Updated on 30th April 2014. Company Confidential.

[37] Janne Viitala and Tero Piispala. MCon user manual. Technical report, Sandvik Mining and Construction. Revision 1.1, Updated on 18th August 2011. Company Confidential.

[38] Janne Viitala and Olli Snellman. SICA – concept level architecture specification. Technical report, Sandvik Mining and Construction. Revision 0.29, Updated on 5th October 2011. Company Confidential.